

An Implementation of J

Roger K.W. Hui

J Version 4.1, 1992 1 27

**Copyright © 1992
Iverson Software Inc.
33 Major Street
Toronto, Ontario
Canada M5S 2K9**

Preface

J is a dialect of APL freely available on a wide variety of machines. It is the latest in the line of development known as “dictionary APL”. The spelling scheme uses the ASCII alphabet. The underlying concepts, such as arrays, verbs, adverbs, and rank, are extensions and generalizations of ideas in APL360. Anomalies have been removed. The result is at once simpler and more powerful than previous dialects.

This book describes an implementation of J in C. The reader is assumed to be familiar with J and C. J is specified by the *ISI Dictionary of J*, and introductions to the language are available in *An Introduction to J* and *Programming in J*; C is described in *The C Programming Language*.

Why “J”? It is easy to type.

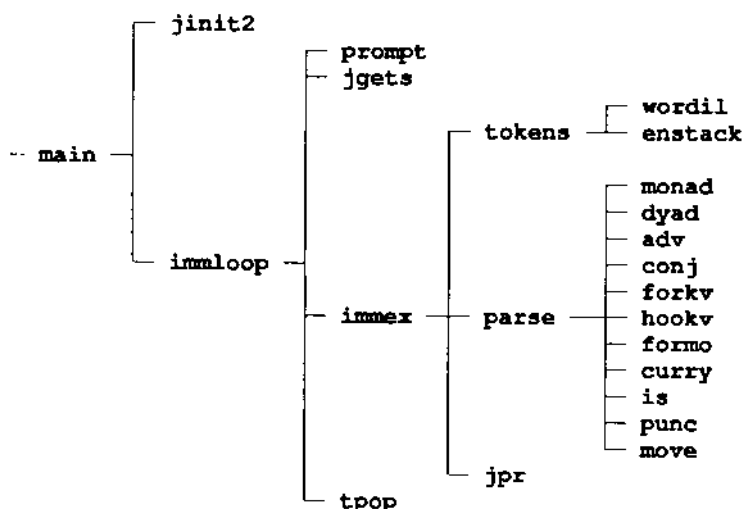
Acknowledgment

Ex ungue leonem.

Contents

- 0. Introduction
- 1. Interpreting a Sentence
 - 1.1 Word Formation
 - 1.2 Parsing
 - 1.3 Trains
 - 1.4 Name Resolution
- 2. Nouns
 - 2.1 Arrays
 - 2.2 Types
 - 2.3 Memory Management
- 3. Verbs
 - 3.1 Anatomy of a Verb
 - 3.2 Rank
 - 3.3 Atomic (Scalar) Verbs
 - 3.4 Obverses, Identities, and Variants
 - 3.5 Error Handling
- 4. Adverbs and Conjunctions
- 5. Representation
 - 5.1 Atomic Representation
 - 5.2 Display Representation
 - 5.3 String Representation
 - 5.4 Tree Representation
- 6. Display
 - 6.1 Numeric Display
 - 6.2 Boxed Display
 - 6.3 Formatted Display
- 7. Comparatives
- 8. Primitives
- Appendices
 - A. Incunabulum
 - B. Program Files
 - C. The LinkJ Interface
 - D. Compiling
 - E. Foreign Conjunction
 - F. System Summary
- Bibliography
- Glossary and Index

0. Introduction



The system is organized as above. The main function `main` calls `jinit2` for initializations, then `immloop` ("immediate execution" loop), which repeats the following steps:

`prompt` and `jgets` prompt and accept an input sentence.

`immex` is the heart of the execution loop. The argument is a string of the input sentence. The processing is divided into three parts:

- `tokens` — word formation — applies the rhematic rules to partition the sentence into words. The result is a list of parts of speech: nouns, verbs, adverbs, conjunctions, copulae, and punctuation.
- `parse` interprets the sentence according to the parsing rules. Parsing is controlled by a table of (pattern,action) pairs; the eleven possible actions are embodied as the functions listed under `parse` in the diagram.
- `jpr` displays the result of the sentence.

Finally, `tpop` frees the temporary storage used in an iteration.

The fundamental data structure is the APL array (an object of data type **A**), used to represent all the possible objects in J. Most functions in the implementation accept arrays as argument and return them as result. Functions tend to be short and compact, and functions which implement J primitives are used freely. Extensive use is made of C preprocessor definitions and macros. Although the implementation language is C, the programming style is unmistakably APL.

The book is organized along the lines of the dictionary: Chapter 1 describes the interpretation of a sentence. Chapters 2, 3, and 4 describe nouns, verbs, and adverbs and conjunctions. Chapter 5 presents alternative representations. Chapter 6 describes display. Chapter 7 describes comparisons. Chapter 8, the final chapter, discusses each primitive in detail.

The remainder of the book contains various useful bits. In particular, Appendix F (on the back cover) provides a means of quickly locating a primitive in the program files, and the Glossary has a short description on every non-local name in those files.

1. Interpreting a Sentence

1.1 Word Formation

Words are expressed in the standard ASCII alphabet. Primitive words are spelled with one or two letters; two letter words end with a period or a colon. The entire spelling scheme is shown on the back cover. The verb `::` facilitates exploration of the rhematic rules. Thus:

```
:: 'sum =. +/_6.95*i.3 4'
```

| | | | | | | | | |
|-----|----|---|---|-------|---|----|---|---|
| sum | =. | + | / | _6.95 | * | i. | 3 | 4 |
|-----|----|---|---|-------|---|----|---|---|

The source code for word formation is in file `w.c`. The process is controlled by the function `wordil` (word index and length) and the table `state`. Rows of `state` correspond to 10 states; columns to 9 character classes. Each table entry is a (new state, function) pair. Starting at state `b`, a sentence is scanned from left to right one character at a time; the table entry corresponding to the current state and character class is applied.

| NEW STATE/FUNCTION | STATES | CLASSES |
|---------------------------|----------------------|--------------------|
| b ?= a= N= a= 9= ?= ?= '= | b Blank | b Blank |
| b> ?> a> N> a> 9> ? ? '> | ? Other | ? Other |
| b> ?> a a a a ? ? '> | a Name | a Letters excl. NB |
| b> ?> a a B a ? ? '> | N N | N The letter N |
| b> ?> a a a a C ? '> | B NB | B The letter B |
| z z z z z z ? ? z | C NB. | 9 Digits and _ |
| b> ?> 9 9 9 9 9 ? '> | 9 Number | . Period |
| ' ' ' ' ' ' ' ' " | ' Quote | : Colon |
| b> ?> a> N> a> 9> ?> ?> ' | " Adjacent Qtes | ' Quote |
| z z z z z z z z z | z Comment | |
| b ? a N B 9 . : ' | FUNCTION | |
| | > j=.i [Emit(j,i-1) | |
| | = j=.i | |

Emit(j, i-1) produces a pair of indices delimiting a word in the string. **i** is the current index, and **j** is an internal register; if the current word is a number immediately following a numeric list (one or more numbers), **Emit** combines their indices to form a single word. At the end of the string, **Emit(j, i-1)** is executed.

This process is applied to `sum =. +/_6.95*i.3 4`, the sentence used above to illustrate word formation. In the following table, the columns are: index, character in the string, the (current state, character class) pair, the (new state, function code) pair, and the action. For example, the first step is step 0, the letter is `s`, the current (and initial) state is `b`, and the character class is `a`. From the table, the entry in row `b` and column `a` is `a=`, meaning the new state is `a` and the function code is `=`. The action assigns 0 to `j`.

| i | CHAR | STATE/ CHAR CLASS | NEW STATE/ FUNCTION | ACTION |
|----|------|----------------------|------------------------|---------------------|
| 0 | s | b a | a= | j=.0 |
| 1 | u | a a | a | |
| 2 | m | a a | a | |
| 3 | | a b | b> | j=.3 [Emit(0,2) |
| 4 | = | b ? | ?= | j=.4 |
| 5 | . | ? . | ? | |
| 6 | + | ? ? | ?> | j=.6 [Emit(4,5) |
| 7 | / | ? ? | ?> | j=.7 [Emit(6,6) |
| 8 | _ | ? 9 | 9> | j=.8 [Emit(7,7) |
| 9 | 6 | 9 9 | 9 | |
| 10 | . | 9 . | 9 | |
| 11 | 9 | 9 9 | 9 | |
| 12 | 5 | 9 9 | 9 | |
| 13 | * | 9 ? | ?> | j=.13 [Emit(8,12) |
| 14 | i | ? a | a> | j=.14 [Emit(13,13) |
| 15 | . | a . | ? | |
| 16 | 3 | ? 9 | 9> | j=.16 [Emit(14,15) |
| 17 | | 9 b | b> | j=.17 [Emit(16,16) |
| 18 | 4 | b 9 | 9= | j=.18 |
| 19 | | | | Emit(18,18) |

Every primitive word has an ID (a unique byte value) defined in file `jc.h`. The ID for the first 128 ASCII characters are simply the byte values 0 to 127; other IDs are arbitrary assignments in “dictionary order”.

```
...
#define CLPAR    '('          /* 40 050 28      */
#define CRPAR    ')'          /* 41 051 29      */
#define CSTAR    '*'          /* 42 052 2a      */
#define CPLUS    '+'          /* 43 053 2b      */
...
#define CASGN    '\200'       /* 128 200 80 =.   */
#define CGASGN   '\201'       /* 129 201 81 =:   */
#define CFLOOR   '\202'       /* 130 202 82 <.  */
#define CMIN     '\202'       /* 130 202 82 <.  */
#define CLE      '\203'       /* 131 203 83 <:   */
#define CCEIL    '\204'       /* 132 204 84 >.   */
#define CMAX     '\204'       /* 132 204 84 >.   */
#define CGE      '\205'       /* 133 205 85 >:   */
...
```

Using mnemonics such as `CPLUS` and `CASGN` instead of `'+'` and `'\200'` makes the source code more readable and more amenable to automatic manipulation.

The 3-row table `spell` associates letter sequences with IDs. The rows correspond to letters in the range ASCII 32 to 127, those letters inflected by a period, and those letters inflected by a colon; table entries are IDs. Thus:

```
static C spell[3][47]={
    '=',    '<',    '>',    '_',    '+',    '*',    ...,
    CASGN,  CFLOOR, CCEIL,  1,      COR,    CAND,    ...,
    CGASGN, CLE,    CGE,    CUSCO,  CNOR,   CNAND,   ...,
}
```

The first column specifies that `=.` has the ID `CASGN` (assignment) and `=:` the ID `CGASGN` (global assignment).

spell is used by functions **spellin** and **spellout**: given a string (e.g. "=."), **spellin** computes the ID (**CASGN**); given the ID, **spellout** computes the corresponding string. **spellin** also uses the table **nu**, which contains alternative spellings for the “national use” characters.

Using the information computed by **word11**, functions **tokens** and **enstack** transform a string into a list of nouns, verbs, adverbs, conjunctions, etc. The next step is to parse this “tokenized” form of the sentence.

1.2 Parsing

Parsing occurs after word formation and is controlled by function `parse` and table `cases` in file `p.c`. `cases` is a direct translation of the parse table in Section II E of the dictionary:

```
#define EDGE      (MARK+ASGN+LPAR)
#define NOTCONJ   (NOUN+VERB+ADV)
#define RHS       (NOUN+VERB+ADV+CONJ)

PT cases[] = {

  EDGE,          VERB,      NOUN,      ANY,      monad,vmonad,cmonad, 1, 2,
  EDGE+NOTCONJ,  VERB,      VERB,      NOUN,      monad,vmonad,cmonad, 2, 3,
  EDGE+NOTCONJ,  NOUN,      VERB,      NOUN,      dyad, vdyad, cdyad,  1, 3,
  EDGE+NOTCONJ,  NOUN+VERB, ADV,      ANY,      adv,  vadv,  cadv,   1, 2,
  EDGE+NOTCONJ,  NOUN+VERB, CONJ,     NOUN+VERB, conj, vconj, cconj, 1, 3,

  EDGE+NOTCONJ,  VERB,      VERB,      VERB,     forkv,vforkv,cforkv, 1, 3,
  EDGE,          VERB,      VERB,      ANY,      hookv,vhookv,chookv, 1, 2,
  EDGE,          ADV+CONJ,  ADV+CONJ,  ADV+CONJ, formo,vformo,cformo, 1, 3,
  EDGE,          ADV+CONJ,  ADV+CONJ,  ANY,      formo,vformo,cformo, 1, 2,
  EDGE,          CONJ,      NOUN+VERB, ANY,      curry,vcurry,ccurry, 1, 2,

  EDGE,          NOUN+VERB, CONJ,      ANY,      curry,vcurry,ccurry, 1, 2,
  NAME+NOUN,     ASGN,      RHS,       ANY,      is,   vis,   vis,    0, 2,
  LPAR,          RHS,       RPARG,     ANY,      punc, vpunc, vpunc,   0, 2,

};
```

The sentence to be parsed is prefaced with a marker and placed on a queue, and as parsing proceeds words are moved from the right end of the queue onto a stack. The classes of the first four words on the stack are compared to the patterns in columns 0 to 3 of `cases`. The first row matching in all four columns is selected; the action in column 4 is applied to the words on the stack indicated by the inclusive indices in columns 7 and 8, with the result replacing those words. If none of the rows match, the word at the end of the queue is moved onto the stack by the function `move`. Scanning for a matching pattern then begins anew. The process terminates when the queue is empty and none of the rules are applicable. At that time, the stack should have exactly two words: the marker and a noun, verb, adverb, or conjunction; anything else signals syntax error.

This parsing method was first described in Iverson [1983]. The parse table is a compact representation of a large amount of information; it has guided both the evolution of the language and its implementation. The following example illustrates parsing on the sentence $((i.\#y)=i.\sim y)\#y$ where $y = \text{'abc'}$. ($\$$ denotes the marker.)

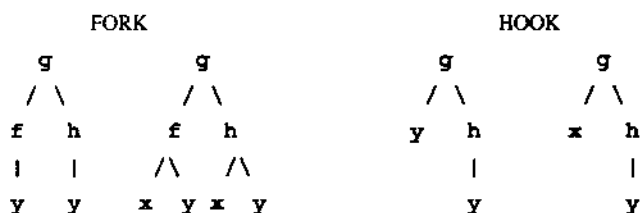
| QUEUE | STACK | RULE/ ACTION | COMMENT |
|--------------------------|--------------------------------------|-----------------|----------------------------|
| $\$(i.\#y)=i.\sim y)\#y$ | | | |
| $\$(i.\#y)=i.\sim y)\#$ | 'aba' 13 | move | |
| $\$(i.\#y)=i.\sim y)$ | #'aba' 13 | move | |
| $\$(i.\#y)=i.\sim y$ |)#'aba' 13 | move | |
| $\$(i.\#y)=i.\sim$ | 'aba')#'aba' 13 | move | |
| $\$(i.\#y)=i.$ | \sim 'aba')#'aba' 13 | move | |
| $\$(i.\#y)=$ | $i.\sim$ 'aba')#'aba' 13 | move | |
| $\$(i.\#y)$ | $=i.\sim$ 'aba')#'aba' 13 | move | |
| $\$(i.\#y)$ | $=v0$ 'aba')#'aba' 3 | adv | $v0=i.\sim$ |
| $\$(i.\#y$ |)= $v0$ 'aba')#'aba' 13 | move | |
| $\$(i.\#$ | 'aba')= $v0$ 'aba')#'aba' 13 | move | |
| $\$(i.$ | #'aba')= $v0$ 'aba')#'aba' 13 | move | |
| $\$($ | $i.\#$ 'aba')= $v0$ 'aba')#'aba' 13 | move | |
| $\$($ | $(i.\#$ 'aba')= $v0$ 'aba')#'aba' 13 | move | |
| $\$($ | $(i.3)=v0$ 'aba')#'aba' 1 | monad | 3 -: #'aba' |
| $\$($ | $(0\ 1\ 2)=v0$ 'aba')#'aba' 0 | monad | 0 1 2 -: i.3 |
| $\$($ | $0\ 1\ 2=v0$ 'aba')#'aba' 12 | punc | |
| $\$($ | $0\ 1\ 2=0\ 1\ 0$)#'aba' 1 | monad | 0 1 0 -: $v0$ 'aba' |
| $\$($ | $(0\ 1\ 2=0\ 1\ 0$)#'aba' 13 | move | |
| $\$($ | $(1\ 1\ 0)$ 'aba' 2 | dyad | 1 1 0 -: $0\ 1\ 2=0\ 1\ 0$ |
| $\$($ | $1\ 1\ 0$ 'aba' 12 | punc | |
| $\$($ | $\$1\ 1\ 0$ 'aba' 13 | move | |
| $\$($ | $\$$ 'ab' 2 | dyad | 'ab' -: $1\ 1\ 0$ 'aba' |

Functions **vmonad**, **vdya**, ... in column 5 of **cases** are used by **vtrans** in file **pv.c** to implement **s** : 11, a tacit verb equivalent to **s** : '' or '' : **s**. As described in Hui, Iverson, and McDonnell [1991], **vtrans** works by parsing **s**. A parallel stack is maintained, and actions on the stack have parallel actions on corresponding objects on the parallel stack. In particular, when an action *applies* a verb to its argument(s), resulting in a noun, the parallel action *composes* the verb with tacit verbs that produce the arguments, resulting in a new tacit verb.

Similarly, functions **cmonad**, **cdya**, ... in column 6 of **cases** are used by **ctrans** in file **pc.c** to implement **s** : 12, a tacit conjunction equivalent to **s** : 2.

1.3 Trains

A *train* is an isolated phrase not interpreted by the parsing rules pertaining to verbs, adverbs, and conjunctions, and (as a matter of language design) may be assigned any meaning whatsoever. Iverson and McDonnell [1989] defined a train of three verbs as a *fork* and a train of two verbs as a *hook*. That is, if *f*, *g*, and *h* are verbs, then so are (*f g h*) and (*g h*), and:



Similarly, trains of two or three adverbs and conjunctions can be assigned meanings. The interpretation of trains of two or three adverbs and conjunctions are as follows:

| TRAIN | RESULT | DEFINITION |
|----------|-------------|----------------------|
| a0 a1 a2 | adverb | x a0 a1 a2 |
| a0 a1 c2 | | undefined |
| a0 c1 a2 | conjunction | (x a0) c1 (y a2) |
| a0 c1 c2 | conjunction | (x a0) c1 (x c2 y) |
| c0 a1 a2 | conjunction | x c0 y a1 a2 |
| c0 a1 c2 | | undefined |
| c0 c1 a2 | conjunction | (x c0 y) c1 (x a2) |
| c0 c1 c2 | conjunction | (x c0 y) c1 (x c2 y) |
| a0 a1 | adverb | x a0 a1 |
| a0 c1 | adverb | x a0 c1 x |
| c0 a1 | conjunction | x c0 y a1 |
| c0 c1 | | undefined |

Finally, a conjunction in isolation with an argument *bonds* (Curries) the argument to the conjunction, producing an adverb.

Parsing rules 5 to 10 deal with trains. (See 1.2 *Parsing*.) A consequence of the rules is that a train of verbs is resolved by repeatedly forming a fork from the *rightmost* three verbs, with a final hook if the train is of even length. Likewise, a train of adverbs and conjunctions is resolved by repeatedly forming a group from the *leftmost* three adverbs or conjunctions, with a final group of two if the train is of even length.

Trains are implemented by the functions and variables in file `ct.c`. The main routines are:

| | |
|----------------|--|
| folk | A train of three verbs (" fork " conflicts with UNIX usage) |
| hook | A train of two verbs |
| forko | A train of three adverbs and conjunctions |
| hooko | A train of two adverbs and conjunctions |
| advform | A conjunction in isolation with an argument |
| gtrain | The noun case of the adverb \ |

1.4 Name Resolution

During parsing, words are moved from the queue to the stack (see 1.2 *Parsing*). Suppose a name **xyz** is being moved. If **xyz** is immediately to the left of a copula, it (as a name) is put on the stack. Otherwise, if **xyz** denotes a noun, that noun is put on the stack; if **xyz** denotes a verb, adverb, or conjunction, '**xyz**'~ is put on the stack, to be evaluated when the verb, adverb, or conjunction is applied.

Names and their assigned values are stored in symbol tables. A symbol table is an array of type **syms** whose atoms are pairs (name,value). (See 2.1 *Arrays*.) Functions and variables in file **s.c** work with symbol tables. In particular, **symsis(a,w,symb)** assigns the name **a** to **w** in the symbol table **symb**, and **symsrd(w)** "reads" the value of the name **w**.

2. Nouns

2.1 Arrays

The fundamental data structure is the array, that is, an object of the C data type **A** defined in file `jt.h`:

```
typedef long I;  
typedef struct {I t,c,n,r,s[1];} *A;
```

All objects, whether numeric, literal, or boxed, whether noun, verb, adverb, or conjunction (or other), are represented by arrays. For example, the string 'Cogito, ergo sum.', the atom 1.61803, and the table 1.3 4 are represented thus:

| | | | | | | | | | |
|------|---|----|---|----|------|-----|------|-----|---|
| | t | c | n | r | s[0] | | | | |
| CHAR | 1 | 17 | 1 | 17 | Cogi | to, | ergo | sum | . |

| | | | | | |
|----|---|---|---|---------|--|
| | t | c | n | r | |
| FL | 1 | 1 | 0 | 1.61803 | |

| | | | | | | | | | |
|-----|---|----|---|---|------|------|---|---|--|
| | t | c | n | r | s[0] | s[1] | | | |
| INT | 1 | 12 | 2 | 3 | 4 | 0 | 1 | 2 | |

| | | | | | | | | | |
|--|---|---|---|---|---|---|---|----|----|
| | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|--|---|---|---|---|---|---|---|----|----|

The parts of an array, and macros for manipulating them, are as follows:

| PART | MACRO | DESCRIPTION |
|------|-------|----------------------------------|
| t | AT | type |
| c | AC | reference count |
| n | AN | number of atoms |
| r | AR | rank |
| s | AS | shape |
| | AV | "value", atoms in ravelled order |

The shape **s** consists of **r** integers whose product equals **n**. The atoms of the array follow immediately after **s**, in ravelled (row major) order. Setting **t**, **n**, **r**, or **s** incorrectly, or exceeding the bounds of the array specified by these quantities, almost always lead to erratic behaviour and catastrophic failure.

The macros **AT**, **AC**, **AN**, and **AR** denote "fullword" integers and may occur on the left or right of an assignment (*i.e.* they are "lvalues"). **AS** is an integer pointer. **AV** is also an integer pointer, and must be *cast* to a C data type appropriate for the type of array. (See 2.2 *Types*.)

All arrays are created using the macro **GA** in file **j.h**. The statement

```
GA(xyz,t,n,r,s);
```

creates an array named **xyz** of type **t** and rank **r**, having **n** atoms and shape **s**. If the rank is 0, **s** is ignored; if the rank is 1, again **s** is ignored, and the shape is set to **n**; otherwise, if **s** is 0, the shape is not initialized by **GA** (and must be initialized subsequently). **GA** returns zero if the array can not be created.

For example, the arrays diagrammed on the previous page might be created as follows, under the names **ces**, **phi**, and **m**:

```
typedef char C;
typedef double D;

A ces,m,phi; I j,*s,*v;

GA(ces,CHAR,17,1,0);
memcpy((C*)AV(ces),"Cogito, ergo sum.",(size_t)17);

GA(phi,FL,1,0,0);
*(D*)AV(phi)=1.61803;

GA(m,INT,12,2,0);
s=AS(m); *s=3; *(1+s)=4;
v=AV(m); for(j=0;12>j;++j) *v++=j;
```

The following utilities in file `u.c` are convenient for creating simple arrays:

| | |
|---------------------------------|---|
| <code>A sc(I k)</code> | An integer atom with value <code>k</code> |
| <code>A scalar4(I t,I k)</code> | An atom of type <code>t</code> with 4-byte value <code>k</code> |
| <code>A scf(D x)</code> | A floating point atom with value <code>x</code> |
| <code>A scc(C c)</code> | A literal atom with value <code>c</code> |
| <code>A apv(I n,I b,I m)</code> | The arithmetic progression vector <code>b+m*i.n</code> . |
| <code>A str(I n,C*s)</code> | A string (literal list) of length <code>n</code> with value the characters pointed to by <code>s</code> . |
| <code>A cstr(C*s)</code> | A string with value the characters in the 0-terminated string <code>s</code> . |

For example, the first two arrays diagrammed on the first page of this chapter might be created by `str(17L,"Cogito, ergo sum.")` or `cstr("Cogito, ergo sum.")` and by `scf(1.61803)`; and `sc(k)` is equivalent to `scalar4(INT,k)`.

A few useful constants are also provided. They are initialized in file `i.c`.

| | |
|-------------------|---|
| <code>zero</code> | <code>0</code> |
| <code>one</code> | <code>1</code> |
| <code>two</code> | <code>2</code> |
| <code>neg1</code> | <code>_1</code> |
| <code>pie</code> | <code>o.1</code> (" <code>pi</code> " conflicts with C usage) |
| <code>a0j1</code> | <code>0j1</code> |
| <code>mtv</code> | <code>\$0</code> |
| <code>jot</code> | <code><\$0</code> |
| <code>dash</code> | <code>'_'</code> |

2.2 Types

If \mathbf{x} is an array, its *type* $\mathbf{AT}(\mathbf{x})$ specifies how the atoms starting at $\mathbf{AV}(\mathbf{x})$ are to be interpreted. In C programming terms, $\mathbf{AV}(\mathbf{x})$ must be *cast* to a pointer of the appropriate C data type:

| $\mathbf{AT}(\mathbf{x})$ | C DATA TYPE | DESCRIPTION |
|---------------------------|----------------|-------------------|
| BOOL | B | Boolean |
| CHAR | C | literal |
| INT | I | integer |
| FL | D | floating point |
| CMPX | Z | complex |
| BOX | A | boxed |
| VERB | V | verb |
| ADV | V | adverb |
| CONJ | V | conjunction |
| NAME | C | name |
| LPAR | I | left parenthesis |
| RPAR | I | right parenthesis |
| ASGN | I | assignment |
| MARK | I | parser marker |
| SYMB | SY | symbol table |

For example, if \mathbf{x} is literal and $\mathbf{s} = (\mathbf{C}^*)\mathbf{AV}(\mathbf{x})$, then $\mathbf{s}[\mathbf{i}]$ is character \mathbf{i} of \mathbf{x} . The C data types in the table are all **typedefs** found in file `jt.h`; the data type **v** is explained in Chapter 4.

Types are fullword integers, and are powers of 2 to permit convenient tests for “composite” types. For example, if:

```
#define NUMERIC      (BOOL+INT+FL+CMPX)
#define NOUN         (NUMERIC+CHAR+BOX)
```

Then the phrase **NUMERIC&AT(x)** tests for numeric arrays, and the phrase **NOUN&AT(x)** tests for nouns. Such comparisons play a key role in the parser (see 1.2 *Parsing*).

A numeric array is accepted as argument by a primitive, regardless of its type, if it is mathematically within the domain of the primitive. For example, a primitive with integral domain would accept integers in an array of type `FL`, `CMPX`, or `BOOL`, or of course `INT`. (This analytic property does not extend to functions internal to the implementation.) Functions in the file `k.c` convert between numeric types. A converted result is an array of the target type equal to the argument within *fuzz* (see 7 *Comparatives*). The following functions are available:

| | |
|------------------------|---|
| <code>cvt(t,x)</code> | Convert <code>x</code> to type <code>t</code> ; signal error if not possible |
| <code>pcvt(t,x)</code> | Convert <code>x</code> to type <code>t</code> ; return <code>x</code> if not possible |
| <code>xcvt(x)</code> | Convert <code>x</code> to the "lowest" type |

The utility `bp` in file `u.c` applies to a type, and returns the number of bytes per atom of that type. Thus `bp(INT)` is 4; `bp(AT(x))` is the number of bytes per atom of `x`; and `16+(4*AR(x))+AN(x)*bp(AT(x))` is the number of bytes required by `x` — 4 bytes each for `t,c,n,x`; 4 bytes each for the `AR(x)` elements of the shape; and `bp(AT(x))` bytes each for `AN(x)` atoms.

The atoms of a boxed array are pointers to other arrays, and are accessible through `(A*)AV(x)`, as the following example illustrates. `aib` applies to a boxed array `x`, and returns the number of atoms in each box of `x`:

```
#define R return

A aib(x)A x; {A*u,z;I j,*v;
  GA(z,INT,AN(x),AR(x),AS(x));          /* 1 */
  u=(A*)AV(x); v=AV(z);                  /* 2 */
  for(j=0;AN(x)>j;++j)*v++=AN(*u++);      /* 3 */
  R z;
}
```

Line 1 creates an integer array `z` having the same rank and shape as `x`. Line 2 initializes pointer variables `u` and `v` for traversing `x` and `z`. Line 3 runs through the atoms of `x`, through `u`, and records the number of atoms in each. Since the data type of `u` is `A*`, the data type of `*u` is `A` and are subject to `AN`, `AT`, `AV`, etc.

2.3 Memory Management

When an array is created, `malloc` is called to obtain the requisite storage; when this storage is no longer needed, `free` is called to return it to the underlying system. No “garbage collection” is done. The performance of this strategy is adequate on modern virtual memory systems. To facilitate the implementation of alternative strategies, the use of `malloc` and `free` is limited to a single instance each, in the file `m.c`.

The reference count of an array is incremented when it is assigned a name, directly or indirectly, and is decremented when the name is re-assigned or erased; when the reference count of an array reaches 0, its storage is freed.

When an array is created, a pointer to it is entered in a “temp stack” (`tstack` in file `m.c`). A *temp* is an array on this stack with a reference count of one. The temp stack plays an important role in the main execution loop (see 0 *Introduction*). In an iteration of the loop,

- The top of the temp stack is recorded;
- A line of user-input is executed; and
- Temps from the current top-of-stack to the old top-of-stack recorded above, are freed.

This device permits functions to be written without explicit memory management code. For example, the monad `, :` is written:

```
F1(lamin1) {R reshape(over(one, shape(w)), w); }
```

And `lamin1` need not be concerned with temps used in `reshape`, `over`, or `shape`, because they are accounted for in the main loop.

On the other hand, a function *may* account for temps: On entry into the function, the current top-of-stack is recorded; on exit, temps are freed down to the recorded point. (These actions are mediated by the macros `PROLOG` and `EPILOG`.) Whether a function accounts for temps does not affect the logic of functions that it calls, nor functions that call it.

3. Verbs

3.1 Anatomy of a Verb

Verbs are implemented as functions. A verb applies to a noun (if used monadically) or to two nouns (if used dyadically), and produces a noun. The data type **AF** and the macros **F1** and **F2** codify these properties:

```
typedef A(*AF) ();  
  
#define F1(f)  A f(w)A w;  
#define F2(f)  A f(a,w)A a,w;
```

AF is the data type of a function having these properties. **F1** and **F2** are used to specify the headers of functions implementing verbs. (They are also used to specify headers of adverbs and conjunctions.) By far the majority of functions in the implementation are so specified. Verbs are represented by arrays of type **VERB**; the details of this representation are deferred until the next chapter, 4 *Adverbs and Conjunctions*.

The verb **j.** is used here to illustrate the relationship between relevant system components. Recall that **j.** has monad **0j1*** and dyad **+j.**, with ranks **_ 0 0**. There are three main steps in the implementation:

1. Define and declare functions which implement the monad and dyad.
2. Associate **j.** with the functions and other information.
3. Specify obverses, identity functions, and variants (if any).

The steps are executed as follows:

1. Functions which implement the monad and dyad **j.** are added to file **vm.c** (or to one of several **v*.c** files), and declarations are added to **je.h**:

FILE **vm.c**

```
F1(jdot1){R times(a0j1,w);}
F2(jdot2){R plus(a,jdot1(w));}
```

FILE **je.h**

```
extern A jdot1();
extern A jdot2();
```

2. The association between `j.` and `jdot1` and `jdot2` is established in the tables `ps` and `psptr` in file `t.c`. `psptr[x]` is the index in `ps` for byte value `x`. The ID for `j.` is `CJDOT` (defined in file `jc.h`; see 1.1 *Word Formation*), therefore the information for `j.` can be found in `ps[psptr[CJDOT]]`. Entries in that locale are as follows:

```
/*199 E. CEBAR */ {VERB, 0,      ebar,  0,  RMAX,RMAX,0  },
/*200 f. CFIX  */ {ADV,  fix,   0,      0,  0,  0,  0  },
/*201 i. CIOTA */ {VERB, iota,  indexof,1,  RMAX,RMAX,0  },
/*202 j. CJDOT */ {VERB, jdot1, jdot2,  RMAX,0,  0,  0  },
/*203 o. CCIRC */ {VERB, pitimes,circle, RMAX,0,  0,  0  },
/*204 p. CPOLY */ {VERB, poly1, poly2,  1,  1,  0,  CPOLY},
```

The entry for `j.` indicates that it is a verb, with monad `jdot1`, dyad `jdot2`, monadic rank `RMAX`, left dyadic rank 0, right dyadic ranks 0, and a non-primitive inverse (if it has an inverse at all). The information in `ps` and `psptr` are used by the utility `ds` ("define symbol") in file `au.c`. `ds` applies to an ID and produces the corresponding primitive. Thus, `ds(CJDOT)` is `j..`

3. A verb may have additional parts which can not be specified as static data structures. (`ps` and `psptr` are static data structures.) Such information is embodied in functions `inv` and `invamp` (obverses) in file `ai.c`, `iden` (identities) in `ai.c`, and `fit` (variants) in `cf.c`. See 3.4 *Obverses, Identities, and Variants*.

The obverse for `j.`, `n&j.`, and `j.&n` are as follows:

```
j.          %&0j1
n&j.        %&0j1@(-&n)  OR  (j.^:_1)@(-&n)
j.&n        -&(j.n)
```

The obverse of `j.` is implemented as `case CJDOT` in `inv`; those for `n&j.` and `j.&n` are implemented as `case CJDOT` in `invamp`. The identity function of `j.` is `$&0@}.@$`, and is implemented as `case CJDOT` in `iden`. `j.` has no variants; the implementation of a variant would have required a `case` in `fit`.

3.2 Rank

The ranks of a verb are three integers of the monadic rank, left rank, and right rank. A verb need only be defined on arguments of rank bounded by its ranks; the extension to higher-ranked arguments is uniform for all verbs. The intrinsic (default) ranks of a verb `u` may be augmented by the rank conjunction, thus: `u"n`, which may be modelled as follows:

```
rank    =. #0$
rep     =. 3&$%.|
cellax  =. 0&>.@(+rank)`(<.rank) @. (0&<:@[])
enl     =. ]`(<0$ , [ $: */@[ ]. ]) @. (*@#0)
enc     =. -@cellax (().$) $ (().$) enl ,@] ]
sfx     =. -@<.&rank
agree   =. (sfx {. $@[] -: (sfx {. $@[]))
frame   =. ('err'&+)`($@([`]@.<.&rank)) @. agree

r       =. rep n
mcell   =. (0{r)&enc
lcell   =. (1{r)&enc@[ [ . lframe =. frame ($,) [
rcell   =. (2{r)&enc@[ [ . rframe =. frame ($,) ]

u"n y    is  u&> mcell y
x u"n y  is  x (lcell (lframe u&> rframe) rcell) y
```

The utility `rank` returns the rank of its argument, and `rep` replicates one or two ranks into three. `r cellax y` computes the number of cell axes for rank `r` and noun `y`; `s enl,y` boxes the first cell of `y` for cell shape `s`; and `r enc y` boxes the cells of `y` for rank `r`. `lcell (rcell)` builds an array of boxed left (right) argument cells; `lframe` and `rframe` check these arrays for agreement (*viz.*, shapes must match in suffix), then reshape the lower-ranked array to the shape of the other. In the expression for the dyad, `u&>` applies to left and right arguments of the same shape; in both expressions, `u` applies to cells with rank bounded by `n`.

(The preceding text borrows extensively from Hui [1987] A.2.)

The model is implemented by functions `ranklex` and `rank2ex` ("rank execution") in file `cr.c`. A function `f` has access to the entire arguments

of the verb that it implements, regardless of the ranks of the verb. Within **f**, rank effects can be achieved by invoking **ranklex** and **rank2ex**, mediated by the macros **F1RANK** and **F2RANK**:

```
A ranklex(    A w,A self,I m,    AF f1);
A rank2ex(A a,A w,A self,I l,I r,AF f2);

F1RANK(m,  f1,self);
F2RANK(l,r,f2,self);
```

a and **w** are the left and right arguments of the verb; **f1** and **f2** are functions which implement the monad and dyad; **m**, **l**, **r** are ranks; and **self** is an array representing the verb (see 4 *Adverbs and Conjunctions*). For example, the dyad **#** has ranks 1 _ and is implemented by the function **repeat**, which uses **F2RANK** as follows:

```
F2(repeat) {A z:C*v,*x;I c,j,k,m,p=0,n,r,t,*u;
  F2RANK(1,RMAX,repeat,0);
  RZ(a=vi(a)); ...
}
```

If the argument ranks are not greater than the verb ranks, then **F2RANK** (**F1RANK**) does nothing, and execution proceeds to the statement following the macro; if the argument ranks *are* greater, then **F2RANK** (**F1RANK**) invokes **rank2ex** (**ranklex**), and on return therefrom exits **f** with the result obtained therefrom. In this scheme, **rank2ex** (**ranklex**) invokes **f** repeatedly, but with arguments of rank bounded by the verb ranks.

A function may implement rank by other means. For example, the dyad **!** has ranks 0 _ and is implemented by the function **from**, which eschews **rank2ex** on numeric left arguments when rank effects are rather simple. (**from** does use **rank2ex** on boxed left arguments.) Atomic verbs also implement rank independently to exploit the special properties of such verbs. See the next section, 3.3 *Atomic (Scalar) Verbs*.

Verbs derived from adverbs and conjunctions are *always* invoked with **self**. The macros **PREF1** and **PREF2** are used in such cases, wherein **ranklex** and **rank2ex** are invoked with ranks extracted from **self**, and not with “hard-wired” numbers as in the use of **F1RANK** and **F2RANK** for primitive verbs.

3.3 Atomic (Scalar) Verbs

An atomic verb is a primitive verb of the form $f''_ : g$ (that is, a verb whose monad is $f''_$ and whose dyad is g), where f and g have zero argument and result ranks. (These are the *scalar functions* in APL.) The *shape* of the result is therefore determined by the shapes of the arguments alone: For monads, the shape of the result is simply the shape of the argument; for dyads, it is the shape of the higher-ranked argument (and the shape of the other argument must be a suffix of this shape). The *type* of the result is determined by the types of the arguments.

Mechanisms described in the previous section (3.2 *Rank*) suffice to implement atomic verbs. However, the special properties of atomic verbs can be exploited to effect more efficient computation, as follows:

In the implementation, the definition of an atomic verb begins by specifying the computation on atoms of each data type, in the form of *kernels*. A kernel is a function defined by the macros **SF1** or **SF2** (in file v.h). For example, the kernels for the dyad `+` are as follows (in file ve.c):

```
static SF2(bplus,B,I, *u+*v)
static SF2(iplus,I,D, *u+(D)*v)
static SF2(dplus,D,D, *u+*v)
static SF2(jplus,Z,Z, zplus(*u,*v))
```

As the examples illustrate, **SF2** has four arguments, f , T_v , T_x , and exp . f is the name of the function being defined; T_v is the data type of the arguments; T_x is the data type of the result; and exp is an expression for computing the result from the arguments, wherein the left argument is available as a pointer of data type T_v named u and the right argument a pointer of data type T_v named v . The definition of the macro is rather shorter than the preceding description:

```
#define SF2(f,Tv,Tx,exp) \
    B f(u,v,x)Tv*u,*v;Tx*x;{*x=(exp); R!jerr;}
```

The formal result of a kernel (*i.e.* the result as far as C is concerned) is Boolean, and is 1 if no errors are encountered. (The variable `jerr` is explained in Section 3.5 *Error Handling*.)

`sf1` is similarly defined. In the expression `exp`, the right (and only) argument is available as a pointer of data type `Tv` named `v`.

The logic for applying kernels is embodied in functions `sex1` and `sex2` ("scalar execution") in file `cr.c`, with the following prototypes:

```
A sex1(    A w,I zt,SF f1)
A sex2(A a,A w,I zt,SF f2)
```

`a` and `w` are the usual array arguments of a verb; `zt` is the type of the result (`BOOL`, `INT`, `FL`, etc.); and `f1` and `f2` are kernels. `sex1` and `sex2` first allocate space for the result, then invoke `f1` and `f2` repeatedly with pointers to the arguments and result.

The definition of an atomic verb is completed by specifying a "cover" function which first coerces the arguments to the same type (or to some type depending on the arguments), then invokes `sex1` or `sex2` with the appropriate result type and kernel. Thus, `+` is implemented by `plus`:

```
F2(plus) {
    switch(coerce2(&a,&w,BOOL)) {
        case BOOL: R sex2(a,w,INT ,bplus);
        case INT:  R pcvt(INT,sex2(a,w,FL,iplus));
        case FL:   R sex2(a,w,FL ,dplus);
        case CMPX: R sex2(a,w,CMPX,jplus);
        default:   R 0;
    }
}
```

`plus` is the function put into the table `ps` in file `t.c`, as described in Section 3.1.

3.4 Obverses, Identities, and Variants

Verbs have additional parts — obverse, identity, and variants — which can not be specified as static data structures. Such information is embodied in functions.

• Obverses

A verb u is an obverse (usually the inverse) of a verb v if $x=u \ v \ x$ for a significant subdomain of v . The obverse is used in the conjunctions *under* ($\&.$) and *power* ($^{\cdot}$). For example, exponential \wedge and logarithm \wedge^{\cdot} are obverses, and:

$$\begin{array}{ll} 3+\&.\wedge.4 & \text{is } \wedge^{\cdot}(\wedge.3)+\wedge.4 & \wedge^{\cdot}\wedge^{\cdot}:_1 & \text{is } \wedge^{\cdot} \\ 3*\&.\wedge.4 & \text{is } \wedge^{\cdot}(\wedge.3)*\wedge.4 & \wedge^{\cdot}\wedge^{\cdot}:_1 & \text{is } \wedge^{\cdot} \end{array}$$

Obverses are produced by the function `inv` in file `ai.c`. (`inv` implements $\wedge^{\cdot}:_1$.) The logic is a combination of table look-up and nested branch tables (`switch-es`).

PRIMITIVES. If the obverse of a primitive verb is itself primitive, the information is recorded in the table `ps` in file `t.c`. For example, the ID for \wedge is `CEXP` and that for \wedge^{\cdot} is `CLOG`; therefore `ps[CEXP].inv` is `CLOG` and `ps[CLOG].inv` is `CEXP`. (`ps[]`.inv is zero otherwise.)

BONDED VERBS. Bonding (Currying) is fixing an argument of a dyad to derive a monad: $n\&v$ or $v\&n$. For example, $10\&\wedge^{\cdot}$ is *base-10 log* and $\wedge^{\cdot}\&0.5$ is *square root*. The obverse of a bonded verb is computed by the subfunction `invamp` in file `ai.c`, invoked by `inv` as appropriate.

PREFIX AND SUFFIX. Sum prefix $+\backslash$ and sum suffix $+\backslash.$ can be expressed as pre-multiplication by matrices obtained by applying $+\backslash$ and $+\backslash.$ on the identity matrix. The obverse is therefore pre-multiplication by the *matrix inverse* of these matrices. (The actual obverse is a more efficient equivalent derived therefrom.) Similar reasoning applies to $-$, $*$, and $\%$, and to $=$ and \sim : on Boolean arguments. The logic is embodied as a sub-`switch` in `inv`, under `case CBSLASH` and `case CBSDOT`.

VERBS DERIVED FROM ~. The monad $\nabla\sim$ computes $y \nabla y$. For example, $+\sim$ is *double*. The obverses of a few such verbs are implemented by a sub-**switch** in **inv**, under case **CTILDE**.

ASSIGNED OBVERSE. A verb may be assigned an obverse with the *obverse conjunction* ($:\cdot$). $f=.u :\cdot \nabla$ is like u but its obverse is ∇ .

OTHER VERBS. **inv** applies to a few other verbs, including $u@v$ and $u\&v$, whose obverse are $(\nabla \text{ inv})@(u \text{ inv})$ and $(\nabla \text{ inv})\&(u \text{ inv})$.

DEFAULT OBVERSE. Verbs which would otherwise be non-obvertable are assigned an obverse $\&.@:\nabla@ (= @i.@\#) +/ . *]$ (function **invdef** in **ai.c**). The reasoning is similar to that under **PREFIX** AND **SUFFIX**.

• Identities

u/y applies the dyad u between the items of y . When y has *zero* items, the result of u/y obtains by applying the *identity function* of u to y , so-called because $u/(iu \ y), y$ or $u/y, (iu \ y)$ is y for a significant subdomain of u .

Identity functions are computed by the function **iden** in file **ai.c**. **iden** behaves like an adverb, applying to verbs and producing verbs. The logic is implemented as a branch table (a **switch**). Not all verbs have identity functions; **iden** signals error in such cases (*i.e.* in $u/''$ when u does not have an identity).

• Variants

Variants of a verb are produced by the *fit* conjunction ($!\cdot$), and are used to effect tolerant comparison ($= < <. <: > >. >: +. * *. -: \sim. \sim: | \# : e. i.$), formatting to a specific precision ($":$ and $5!:3$), shifts ($| \cdot$), and factorial polynomials (\wedge).

$!\cdot$ is implemented by the function **fit** in file **cf.c**. The logic is implemented as a branch table (a **switch**). Not all verbs have variants; **fit** signals error in such cases.

3.5 Error Handling

When an error is encountered in a function, the global variable `jerr` is set to an error number, and zero is returned. Therefore, when calling a function that can not have zero as a valid result (but does return a result), the returned value must be checked for zero; when calling a “void” function or one whose range includes zero, `jerr` must be inspected.

Error numbers range between 1 and `NEVM`, and are referenced by the `ev*` names (“event” names, defined in file `j.h`). The function `jsignal` (u.c) applies to an error number, sets `jerr` to this number, and (if the global variable `errsee` is 1) displays the appropriate error message; `jsignal` exits immediately if `jerr` is already nonzero. `qevm` is a list of the error messages. These messages are initialized in function `jinit2` (i.c), and may be inspected and changed by the user through `9!:8` and `9!:9`.

The macro `ASSERT` (`j.h`) is used extensively in argument validation. It applies to a proposition and an error number. For example, the following statements check whether `w` is a literal atom:

```
ASSERT(!AR(w), EVRANK);
ASSERT(CHAR&AT(w), EVDOMAIN);
```

If the proposition is nonzero, execution proceeds to the next statement; otherwise, the indicated error is `jsignal`-ed and a zero is returned. The macros `RZ` and `RE` (`j.h`) are used in function calls. `RZ` returns zero if its argument is zero; `RE` evaluates its argument, and returns zero if `jerr` is nonzero. For example, the function `iota` (implementing the monad `i.`) uses `RZ` to check the results of functions that it calls, as follows:

```
F1(iota) {A z; I m,n,*v;
  F1RANK(1,iota,0);
  RZ(w=vi(w));
  n=AN(w); v=AV(w); m=prod(n,v);
  RZ(z=reshape(mag(w),apv(ABS(m),0L,1L)));
  DO(n*!m, if(0>v[i])RZ(z=ranklex(z,0L,n-i,reverse)));
  R z;
}
```

The arguments of a function may be the result of another function; the convention is that a function checks its arguments for zero and returns zero immediately in such cases. Thus, in *iota* above:

```
RZ (z=reshape (mag (w) , apv (ABS (m) , 0L, 1L) ) ) ;
```

If **reshape** did not check for zero arguments, the statement would have to be elaborated:

```
RZ (t0=mag (w) ) ;
RZ (t1=apv (ABS (m) , 0L, 1L) ) ;
RZ (z=reshape (t0, t1) ) ;
```

A *conventional function* is a function that follows the conventions described herein — return zero on zero arguments and on errors. The data type **AF** (jt.h) typifies a conventional function. Most functions in the system are conventional; in particular, all functions implementing primitives are conventional. Expressions and statements that use only conventional functions need not employ **RZ** or **RE**, and the resulting programs are neater. For example, consider the functions **lamin1** and **lamin2** (vs.c), implementing *laminare* (: .):

```
F1(lamin1) (R reshape (over (one, shape (w) ) , w) ; )
F2(lamin2) (RZ (a&&w) ; R over (AR (a) ?lamin1 (a) : a,
                                AR (w) ?lamin1 (w) : AR (a) ?w : table (w) ) ; )
```

lamin2 must check for zero arguments **RZ (a&&w)**, because it applies the *unconventional* macro **AR** to the arguments. In contrast, **lamin1** applies only conventional functions to *its* argument and to results of conventional functions on that argument.

4. Adverbs and Conjunctions

An adverb is monadic, applying to a noun or verb argument on its *left*; a conjunction is dyadic, applying to noun or verb arguments on its *left and right*. The result is usually a verb, but can also be a noun, adverb, or conjunction.

The conjunction `&` is used here to illustrate the relationship between relevant system components. (The implementation of adverbs is similar.) Recall that `&` derives a verb depending on whether the arguments are nouns (`m` and `n`) or verbs (`u` and `v`):

```
m&n      undefined
m&v      m&v y is m v y
u&n      u&n y is y u n
u&v      u&v y is u v y; x u&v y is (v x)u(v y)
```

A verb derived from `&` is (internally) an array of type `VERB` whose value is interpreted according to the data type `v`, defined in file `jt.h` as follows:

```
typedef struct {A f1,f2;A f,g,h;I mr,lr,rr;C id;} V;
```

| | |
|--|-----------------------------------|
| <code>f1</code> monad | <code>mr</code> monadic rank |
| <code>f2</code> dyad | <code>lr</code> left dyadic rank |
| <code>f</code> left conj. or adverb argument | <code>rr</code> right dyadic rank |
| <code>g</code> right conj. argument | <code>id</code> identification |
| <code>h</code> auxiliary argument | |

If `fn=.%.&|:`, the internal array for `fn` is:

| t | c | n | r |
|------|---|---|---|
| VERB | 1 | 1 | 0 |

| | | | | | | | |
|-----|-----|------|---|---|----|----|-------|
| on1 | on2 | %. : | 0 | — | — | — | & |
| f1 | f2 | f | g | h | mr | lr | rr id |

Access to fields in `fn` is by name and by macros defined in `jt.h` and `a.h`, and *never* by offsets and indices. Thus, `AV(fn)` points to the "value" of `fn`; and if `v=(V*)AV(fn)`, then `v->f1` is `on1`; `v->f2` is `on2`; `v->f` is the array for `*.`; `v->g` is the array for `|`; (that is, `v->f` and `v->g` are arrays similar to `fn`); `v->mr` is `_` (indicating that `fn` has infinite monadic rank); and so on. The macro `VAV(f) — ((V*)AV(f)) —` is useful for working with adverbs and conjunctions.

To introduce `&` into the system, functions which implement `&` are added to file `c.c` (or to one of several `c*.c` files), and declarations of global objects are added to file `je.h`:

FILE `c.c`:

```
static DF1(withl) (DECLFG; R g2(fs,w,gs);)
static DF1(withr) (DECLFG; R f2(w,gs,fs);)
static CS1(on1, f1(g1(w,gs),fs))
static CS2(on2, f2(g1(a,gs),g1(w,gs),fs))

F2(amp) {
  RZ(a&&w);
  switch(CONJCASE(a,w)) {
    case NN: ASSERT(0,EVDOMAIN);
    case NV: R CDERIV(CAMP,withl,0L,RMAXL,RMAXL,RMAXL);
    case VN: R CDERIV(CAMP,withr,0L,RMAXL,RMAXL,RMAXL);
    case VV: R CDERIV(CAMP,on1,on2,mr(w),mr(w),mr(w));
  })
}
```

FILE `je.h`:

```
extern A amp();
```

Corresponding to the four possibilities, `amp` defines four cases, which either signal error or return a verb; the functions `withl`, `withr`, `on1`, and `on2` are invoked when a verb derived from `&` is applied. For example, `*.&|: m=.?4 4$100` first branches to the case `vv` in `amp`, and subsequently applies `on1` to `m`. Consider a partial macro expansion of `on1` and the values of its local variables for this example:

MACRO EXPANSION:

```
static A on1(w,self)A w,self; {PROLOG;V*v=VAV(self);
  A fs=v->f; AF f1=fs?VAV(fs)->f1:0, f2=fs?VAV(fs)->f2:0;
  A gs=v->g; AG g1=gs?VAV(gs)->f1:0, g2=gs?VAV(gs)->f2:0;
  PREF1(on1);
  z=f1(g1(w,gs),fs);
  EPILOG(z);
}
```

LOCAL VARIABLES:

| | | | |
|-------------|-----------|--|---------------------|
| w | m | | |
| self | fn | above | |
| v | | pointer to the value part of the array fn | |
| fs | % | f1 | monad of % . |
| gs | | g1 | monad of : |
| | | f2 | dyad of % . |
| | | g2 | dyad of : |

The initialization of **v**, **fs**, **f1**, and so on are the same for all adverbs and conjunctions. (The details of such initialization are normally suppressed by the use of macros.) If an argument to **&** (i.e. **fs** or **gs**) is itself a result of adverbs and conjunctions, expressions such as **g1(w,gs)** or **f1(xx,fs)** engender further executions as occurs in **on1**. The macro **PREF1** implements rank (see 3.2 *Rank*), and the macros **PROLOG** and **EPILOG** manage memory (see 2.3 *Memory Management*).

The association between **&** and **amp** is established in the tables **ps** and **psptr** in file t.c. **psptr[x]** is the index of the entry in **ps** for byte value **x**. The ID for **&** is **CAMP** (defined in file jc.h; see 1.1 *Word Formation*), so **ps[psptr[CAMP]]** contains the information for **&**:

```
/* 38 26 & CAMP */ {CONJ, 0, amp, 0, 0, 0, 0 },
```

The entry specifies that **&** is a conjunction and has monad 0 (none), dyad **amp**, ranks 0, and inverse 0 (none). The information in **ps** and **psptr** are used by the utility **ds** ("define symbol") in file au.c. **ds** applies to an ID, and produces the corresponding primitive. Thus, **ds(CAMP)** is **&**.

The utilities `ac1` and `ac2` in file `au.c` enable non-primitive functions (those *not* put into `ps` and `psptr`) to participate in phrases involving adverbs and conjunctions. Suppose `f1` and `f2` are functions which apply to array arguments and return array results. That is, the prototypes of `f1` and `f2` are:

```
A f1(A w);
A f2(A a, A w);
```

Then `ac1(f1)` is a monadic verb and `ac2(f2)` is a dyadic verb, and are in the domain of adverbs and conjunctions. These verbs have infinite ranks; other ranks can be specified through the function `qq` (which implements `"`). Thus, `qq(ac1(f1), sc(1L))` is a verb with rank 1. An ambivalent verb obtains by a further application of the function `colon` (which implements `:`). Thus: `colon(ac1(f1), ac2(f2))` is a verb whose monad is `ac1(f1)` and whose dyad is `ac2(f2)`.

The utilities `df1` and `df2` in file `au.c` apply the monad or the dyad of a verb. For example,

```
df1(w, ds(CPOUND))
df1( w, amp(ds(CPOUND), ds(COPE)))
df2(a, w, amp(ds(CPOUND), ds(COPE)))
df1(w, qq(ac1(f1), sc(1L)))
```

The phrase `ds(CPOUND)` is the verb `#`, `ds(COPE)` is the verb `>`, and `amp(ds(CPOUND), ds(COPE))` is `#&>`; the examples compute `#w`, `#&>w`, `a#&>w`, and `f1` on the lists of `w`. Finally, `df1(w, ac1(f1))` is equivalent to `f1(w)`, and `df2(a, w, ac2(f2))` is equivalent to `f2(a, w)`.

5. Representation

5.1 Atomic Representation

`5! : 1` is a verb that applies to a boxed name, and produces the *atomic representation* of the named object. Gerunds (results of the ``` conjunction) are arrays of atomic representations. The adverb `5! : 0` defines an object from its representation.

The atomic representation is a boxed list of two boxes:

| | | |
|-------------|----|-----------|
| noun | ID | value |
| verb | ID | arguments |
| adverb | ID | arguments |
| conjunction | ID | arguments |

The ID is a string computed by the function `spellout` in file `w.c`. For a primitive with an assigned word (for example `+` or `/.`), the ID is simply that word; for those without, the ID is one of the following:

| | |
|-----|------------------------------|
| '0' | noun |
| '2' | hook |
| '3' | fork |
| '4' | bonded conjunction |
| '5' | 2-element a-train or c-train |
| '6' | 3-element a-train or c-train |

The “value” in the representation of a noun is just the noun itself; arguments in the representation of a verb, adverb, or conjunction are themselves atomic representations. If an object is uniquely identified by the ID alone, then the second field is elided, and the representation is the boxed ID alone.

The following examples illustrate atomic representation:

```
ar = . 5! : 1
```

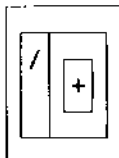
```
plus = . +
```

```
ar < 'plus'
```



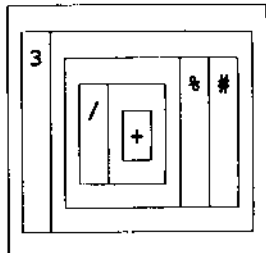
```
sum = . +/
```

```
ar < 'sum'
```

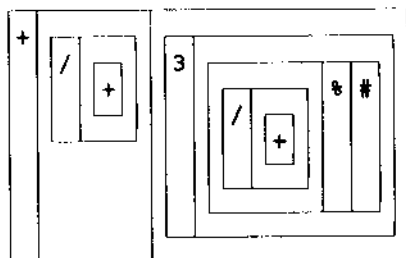


```
mean = . +/ % #
```

```
ar < 'mean'
```

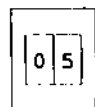


```
+` (+/)` (+/ % #)
```



```
a = . 5
```

```
ar < 'a'
```

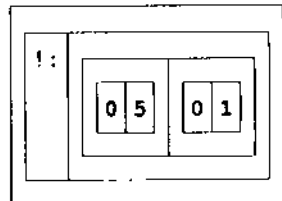


```
xenos = . ! :
```

```
ar < 'xenos'
```

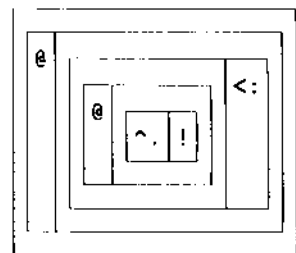


```
ar < 'ar'
```



```
lngamma = . ^ . @ ! @ < :
```

```
ar < 'lngamma'
```



5.2 Display Representation

5!:2 is a verb that applies to a boxed name, and produces the *display representation* of the named object. (This is what is displayed if the result of an input line is a verb, adverb, or conjunction.) The representation can be modelled as follows:

```
ar      =. 5!:1
type    =. 3!:0
boxed   =. 32&=@type
oarg    =. >@ (1&{)

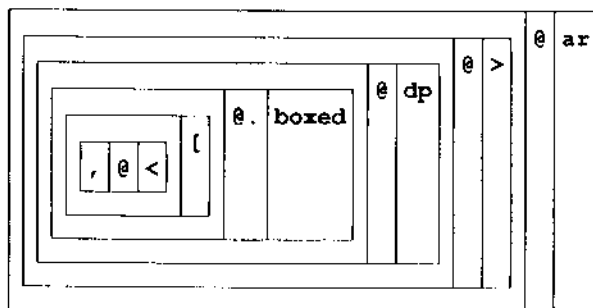
root    =. (<1 0)&C.@,` ] @. (e.&(&.>'0123456789')@[])

dpx     =. {.root dp&.>@oarg
dpogl   =. {.root (dpx&.>@{. , dp &.>@.)@oarg
dpgr     =. {.root (dp &.>@{. , dpx&.>@.)@oarg
dpg     =. dpgr`dpogl`dpx @. (i.&(<,'')@oarg)
dptil   =. dpx` (oarg@>@{.@oarg) @. ((<,'0')&=@{.@>@{.@oarg)
dpcase  =. oarg`dpogl`dpogl`dpg`dptil`dpx @.
                                     ((: '0@.' :4~')&i.@{. )

dp      =. ]`dpcase @. boxed

display =. ,@<`[@.boxed @ dp @ > @ ar
```

display <'display'



The model is divided into groups of verbs. The first group are utilities:

| | |
|--------------|--|
| ar | atomic representation |
| type | type |
| boxed | 1 if boxed |
| oarg | open the second element of the list argument |

root produces an infix representation from a root **r** and its list of arguments **a**. If **r** is a digit, it denotes a primitive without an assigned word (e.g. '3' denotes a *fork*; see 5.1 *Atomic Representation*), and the result of **root** is **a**; otherwise, **r root a** produces:

| | |
|-----------------------|--------------------------|
| a, r | one argument |
| ({.a), r, {.a} | two arguments |
| r | no arguments (primitive) |

The verbs named with the **dp** prefix apply to the opened atomic representation, and embody logic to effect "nice" displays for various special cases. The agenda items in **dpcase** are:

| ID | AGENDA | |
|-------|--------------|---|
| 0 | oarg | noun (leaf) |
| @. | dpgl | gerundial left subtree |
| `: | dpgl | gerundial left subtree |
| 4 | dpq | bonded conjunction; gerundial left or right subtree |
| ~ | dptil | possible instance of <i>evoke</i> |
| other | dp* | none of the above |

display is a model of 5!:2.

5.3 String Representation

`5! :3` is a verb that applies to a boxed name, and produces a literal list of the *string representation* of the named object. The representation conforms to the Workspace Interchange Standard (Bernecky *et al.* [1981]), and facilitates exchange of data and programs between disparate systems.

```
sr =. 5! :3
str =. 'Cogito, ergo sum.'
sr < 'str'
27cstr 1 17 Cogito, ergo sum.
```

```
] ces =. ;: str
```

| | | | |
|--------|---|------|------|
| Cogito | , | ergo | sum. |
|--------|---|------|------|

```
sr < 'ces'
60xbces 1 4 13c- 1 6 Cogito8c- 1 1 ,11c- 1 4 ergo11c- 1
4sum.
```

```
sum =. +/
sr < 'sum'
38xvsum 1 2 8c- 1 1 /17xb- 1 1 8c- 1 1 +
```

The string representation is the catenation of the following parts:

length Digits representing the length of the representation (excluding the length itself).

type One or two letters denoting the type of object

c literal (character) array

n numeric array

xb boxed array

xv verb

xa adverb

xc conjunction

The representation of a verb, adverb, or conjunction is the representation of its opened atomic representation.

name The name of the object, or - if anonymous.

| | |
|-----------------|---|
| <i>blank</i> | A single blank. |
| <i>rank</i> | Digits representing the rank. |
| <i>blank</i> | A single blank. |
| <i>shape</i> | Digits and blanks representing the shape, terminating in a blank. |
| <i>elements</i> | The ravelled elements. For a literal or numeric array, this is the display of the ravelled array; for a boxed array (hence for a verb, adverb, or conjunction), the elements themselves are recursively so represented. |

String representation can be modelled by the following verbs:

```

ar    =. 5!:1
type  =. 3!:0
boxed =. 32%=@type
nc    =. 4!:0

nt    =. >@({&(:'n c n n n xb'))@ (1 2 4 8 16 32%i.)@type
rs    =. (' '&,)@ (,&' ' )@":@ ($@$, $)
elem  =. (":@,)' (;@ ('-'&sn%.>"1)@, )@.boxed
sn    =. {,~ ":@#) @ (nt@],>@[, (rs,elem)@])
st    =. ('x'&,@ ({&' vac')@nc)' (nt@" .@>)@. (2%=@nc)
val   =. (>@ar)' (".&>)@. (2%=@nc)
upfx  =. } ,~ >:@ (<./)@ (i.&'cnb')
sr    =. {,~ ":@#) @ (st , ] upfx@sn val)

```

The first group are utilities:

```

ar      atomic representation
type    type
boxed   1 if boxed
nc      name class

```

nt computes the *type* part of the representation for a noun; the result is *n*, *c*, or *xb*, depending on whether the argument is numeric, literal, or boxed.

rs computes *rank* and *shape*; **rs y** is ' ', (": (\$\$y), \$y) ', the formatted result of the rank and shape, surrounded by blanks.

elem computes *elements*. If the argument **y** is open, this is simply ":", **y**, the format of the ravel of **y**, if boxed, it is the catenation of the representations of the boxed elements.

sn computes the representation of a noun whose boxed name is the left argument, and whose value is the right argument. The parts of the representation correspond to readily identifiable phrases in the definition: *length*, ":%#; *type*, **nt**; *name*, >0[; *rank* and *shape*, **rs**; and *elements*, **elem**.

st computes the *type* part of the representation. The argument is a boxed name; the result is **n**, **c**, **xb**, **xv**, **xa**, or **xc**, according on whether the named object is numeric, literal, boxed, verb, adverb, or conjunction.

val computes the value to be represented, given a boxed name. If the named object is a noun, the value is simply the noun itself (execute the open of the boxed name); otherwise it is the opened atomic representation.

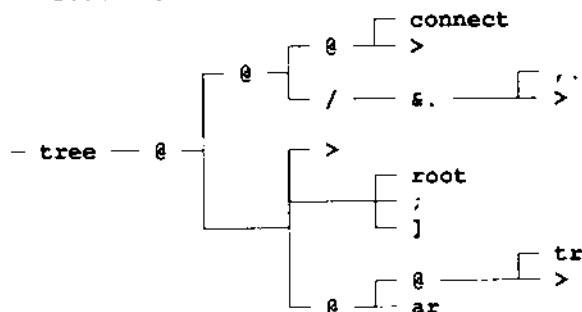
upfx, "unprefix", drops the *length* and *type* parts of the representation of a noun. The amount to be dropped is one plus the minimum index of **c**, **n**, or **b** in the argument.

sr is a model of 5!:3.

5.4 Tree Representation

5!:4 is a verb that applies to a boxed name, and produces a literal table of the *tree representation* of the named object.

```
tree =. connect @ > @ (, .&. >/) @ (> (root;]) tr@>@ar)
5!:4 <'tree'
```



```
ar      =. 5!:1
type    =. 3!:0
boxed   =. 32&=@type
mt      =. 0&@.@$
oarg    =. >@{1&{)
shr     =. |.!. ''
shl     =. 1&{|.!. ''
mat     =. {1 1&}.)@(_1 _1&}.)@"":@<
boxc    =. 9!:6 ''
dash    =. 10{boxc

extent  =. (+./\ *. +./\.) @ (' '&~:) @: ({."1)
limb1   =. 1&|.@$ 1&~: }. {10 6 0{boxc}&, @($&(9{boxc}))
limb    =. -@{i.&1}@[ |. #@[ { . limb1@]
pfx     =. {limb +/}@extent ,. ]
pad     =. [{.] ,. dash&=@({:"1)@] { ' '&, :@($&dash)@(-&(:$)
take    =. pad`({.&{, ' ')}@[] @. {mt@})
rc      =. #@>@{"1 ; >./@:({:@$@>)
kernt   =. {0{boxc}&=@shl@[ *. ' '&~:@]
```

```

kernb  =. (6{boxc)&=0] *. ' '~:0shl0[
kern   =. (<0 0)&{>"2 (kernt+./"10:+.kernb) (<_1 0)&{>"2
gap    =. ,&.>"_1 {&((0 1$' ');1 1$' ')}@kern
graft  =. (pfx&.>@{.})] 0 (,&.>/) 0 gap 0 ({@rc takes.> ])

lab     =. ,: @ (2&|. ) @ ((' ',dash,dash,' ')&,)
label   =. lab`((.dash)&[]) @. (e.&'0123456789'@{.})
center  =. ((i.&1) -@+ <.&-:@(+/))@] |. #0] (. [
root    =. label@[ center extent@>@{.0]

leaf    =. ,@<@(((,:dash,' ')&[ center $&10#) ,. ])@mat@":

trx     =. >@{. (root ; ]) graft@:(tr@>)@oarg
trgl    =. >@{. (root ; ]) graft@:(tr@>@{. , tr @>@{.})@oarg
trgr    =. >@{. (root ; ]) graft@:(tr @>@{. , tr@>@{.})@oarg
trg     =. trgr`trgl`trx @. (i.&(<,' '))@oarg
trtil   =. trx`(leaf@oarg@>@{.0oarg) @.
                                     ((<,'0')&=@{.0>@{.0oarg)
trcase  =. (leaf@oarg)`trgl`trgl`trg`trtil`trx @.
                                     ((:;'00.'~4~')&i. 0{.)
tr       =. leaf`trcase @. boxed

rep     =. [. & (((# i.0#)0,0) (0]))
right   =. (5{boxc) rep (e.&(9{boxc) *. shr"10(e.&dash))
cross   =. (4{boxc) rep (e.&(5{boxc) *. shl"10(e.&dash))
left    =. (3{boxc) rep (e.&(9{boxc) *. shl"10(e.&dash))
bot     =. (7{boxc) rep (e.&(6{boxc) *. shr"10(e.&dash))
connect =. bot @ left @ cross @ right

tree    =. connect @ > @ (,&.>/) @ (> (root;]) tr@>@ar)

```

The model is divided into groups of definitions (which are verbs unless indicated otherwise). The first group are utilities:

```

ar      atomic representation
type    type
boxed   1 if boxed

```


| ID | AGENDA | |
|-------|------------------|--|
| 0 | leaf@oarg | noun (leaf) |
| @. | trgl | gerundial left subtree |
| `: | trgl | gerundial left subtree |
| 4 | txg | bonded conjunction; gerund left or right |
| ~ | trtil | possible instance of <i>evoke</i> |
| other | trx | none of the above |

rep is a conjunction whose left argument is a single literal **c** and whose right argument is a proposition **p**, deriving a verb such that the phrase **c rep p y** replaces with **c** the positions in **y** marked by **p y**.

connect substitutes **⊥** (**bot**), **⊢** (**left**), **⊥** (**cross**), and **⊥** (**right**) at nexuses of the tree.

tree is a model of 5! : 4.

6. Display

If the last operation in a line of user input is not assignment, the result of the line is displayed. More specifically, if the global variable `asgn` is zero at the end of executing an input line, and the line had no errors, `jpr` is invoked to display the result. `jpr` first applies `thorn1` (the monad `" :`) to compute the *display* of `y`, then writes the lines to the screen.

In all cases, the display of an object is a literal array. The display of a literal array is itself. The display of a verb, adverb, or conjunction is that of its display representation `5! : 2` (a boxed array; see Section 5.2). The display of a numeric array is discussed in Section 6.1; that of a boxed array, in Section 6.2; and *format* (the dyad `" :`) is discussed in Section 6.3.

Display is implemented by functions and variables in file `f.c`.

6.1 Numeric Display

The display of a numeric array **y** is a literal array having the same rank as **y** (but at least one), such that the shape of `":y` matches the shape of **y** in all but the last axis. Columns are right-justified and are separated by one space. The conversion from numeric to literal can be modelled as follows:

```
sprintf =. ":
type    =. 3!:0
real    =. {.@+.
imag    =. {.@+.

minus   =. $&'_'@('-'&=@{.)
ubar    =. >@({&(<:._1 ' _ _ _ . _'))@('iInN'&i.@{.)
afte    =. minus , (i.&0@{e.&'-'&0'}) . ]
efmt    =. >:@(i.&'e') ({.,afte@{.}) ]
finite  =. ]`efmt@.('e'&e.)
message =. finite`ubar@.(e.&'iInN'@{.)
fmtD    =. (minus,message@{e.&'-'&+@{. } . ])) @ sprintf

fmtB    =. {&'01'
fmtI    =. sprintf
fmtZ    =. fmtD@real , 'j'&,@fmtD@imag
fmt     =. (fmtB&.>) `(fmtI&.>) `(fmtD&.>) `(fmtZ&.>) @
                                           (1 4 8&i.@type)

sh      =. {*/@:,{:@(1&,))@$ ($, ) ]
width   =. (<:@{. 0} ] )@>:@(>./)@sh@:(#&>)
th      =. (-@width ;@:({.&.>)"1 ] ) @ fmt
```

The model is divided into groups of verbs. The first group are utilities:

| | |
|----------------|--|
| sprintf | a function in the C library |
| type | type |
| real | the real part of a complex number |
| imag | the imaginary part of a complex number |

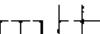
fmtD formats a real number. Its constituents transform the result of **sprintf** to follow J conventions in the treatment of negative signs (**minus**), exponential notation (**efmt** and **afte**), and infinities and indeterminates (**ubar**).

fmt formats a numeric array into an array of boxed strings. It invokes formatters specialized for the different types: **fmtB** (Boolean), **fmtI** (integer), **fmtD** (floating point), and **fmtZ** (complex).

sh shapes an array into a table having the same number of rows. **width** computes the maximum width in each column of an array of boxed strings. **th** is a model of " : " on numeric arrays.

6.2 Boxed Display

The display of a boxed array **b** is a literal array **d=.":b** such that:

- The rank of **d** is the greater of 2 or the rank of **b**.
- Excluding the last two axes, the shape of **d** matches the shape of **b**.
- The frame (formed by ) is the same in all the planes.

Boxed display can be modelled as follows:

```
type      =. 3!:0
boxed     =. 32&= @ type
mt        =. 0&e.@$
boxc      =. 9!:6 ''
tcorn     =. 2 0{boxc
tint      =. 1 10{boxc
bcorn     =. 8 6{boxc
bint      =. 7 10{boxc

sh        =. (*@): , (:)@{1&,)@$ $ ,
rows      =. */\.@):@$
bl        =. ).@{, &0)@{+/)@{0&=)@{|/ i.@{. @{, &1))
mask      =. 1&, . #&, . , &0@>:@i.@#
mat       =. mask@bl@rows { ' ' &, @sh

edge      =. , @{1&, .)@{ } . @# +:@#@[ $ ]
left      =. edge&(3 9{boxc)@>@{0&{)@{ , "0 1 ]
right     =. edge&(5 9{boxc)@>@{0&{)@{ , ~"0 1 ]
top       =. 1&|. @{tcorn&, )@{edge&tint)@>@{1&{)@{ , "2 ]
bot       =. 1&|. @{bcorn&, )@{edge&bint)@>@{1&{)@{ , "2~ ]
perim     =. [ top [ bot [ left [ right ]

topleft   =. {4{boxc}&(0) @ ((_2{boxc)&, .) @ ((_1{boxc)&, )
inside    =. 1 1&|. @ ; @: {, .&.>/"1) @: {topleft&.>)
take      =. [ { . ]`{)&' ' )@.mt@
frame     =. [ perim @{ inside@:(take&.>)"2 ]
rc        =. (>./@sh&.>) @: {, .@|:"2@:(0&{"1);1&{"1) @: ($&>)

thorn1    =. ":`thbox @. boxed
thbox     =. {rc frame ] @: {mat@thorn1&.>}
```

The model is divided into groups of definitions (which are verbs unless indicated otherwise). The first group are utilities:

| | |
|--------------|-------------------------------------|
| type | type |
| boxed | 1 if boxed |
| mt | 1 if empty |
| boxc | (noun) box drawing characters |
| tcorn | (noun) the characters \top \top |
| tint | (noun) the characters \top $-$ |
| bcorn | (noun) the characters \bot \bot |
| bint | (noun) the characters \bot $-$ |

mat is the main verb of the next group of definitions. The argument is a literal array; the result is a literal matrix image of the array — a literal table that “looks like” the argument array.

perim draws a perimeter around each plane of the right argument: According to the information in the left argument (a result of **rc**), **perim** puts \top \top \top $-$ (**top**), \bot \bot \bot $-$ (**bot**), \top \bot (**left**), or \bot \top (**right**) at appropriate positions on the perimeter of each plane.

topleft catenates the characters \top \bot $-$ on the top and left edges of a literal table. **inside** produces the inside (excluding perimeter) of a plane of the display. **take** is $\{.$ if the right argument is non-empty, and is an array of blanks otherwise. **frame** applies to an array of boxed tabular displays, and computes the overall display. **rc** computes the number of rows and columns in the display of the atoms in a plane.

thorn1 models “:”; **thbox** models “:” on a boxed array.

The following examples illustrate the inner workings of the model:

```
y = .(i.2 3); 'abc'; (i.4 1); (<2 2$'ussr'); 12; <+&. > i.2 2 3
y = . 2 3$y
x = .mat@th&.>y
```

\$&.>x

| | | | | | |
|---|---|---|---|----|---|
| 2 | 5 | 1 | 3 | 4 | 1 |
| 4 | 4 | 1 | 2 | 11 | 9 |

rc x

| | | | | |
|---|----|---|---|---|
| 4 | 11 | 5 | 3 | 9 |
|---|----|---|---|---|

{rc x

| | | | | | |
|----|---|----|---|----|---|
| 4 | 5 | 4 | 3 | 4 | 9 |
| 11 | 5 | 11 | 3 | 11 | 9 |

a =. 2 3 4\$'abcdefghijklmnoqrstuvw'

a

abcd

efgh

ijkl

mnop

qrst

uvwx

mat a

abcd

efgh

ijkl

mnop

qrst

uvwx

\$a

2 3 4

\$ mat a

7 4

topleft 3 4\$'a'

```
aaaa
aaaa
aaaa
```

(2 3;4 5) perim 6 10\$'a'

```
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
```

t=.({rc x)inside@:(take&.>)"2 x

t

| | | | | | | | | | | |
|----|----|----|-----|---|---|---|---|---|----|----|
| 0 | 1 | 2 | abc | 0 | | | | | | |
| 3 | 4 | 5 | | 1 | | | | | | |
| | | | | 2 | | | | | | |
| | | | | 3 | | | | | | |
| 12 | | | | <table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>2</td><td>3</td><td>5</td></tr></table> | 0 | 1 | 2 | 2 | 3 | 5 |
| 0 | 1 | 2 | | | | | | | | |
| 2 | 3 | 5 | | | | | | | | |
| us | | | | <table><tr><td>6</td><td>7</td><td>8</td></tr><tr><td>9</td><td>10</td><td>11</td></tr></table> | 6 | 7 | 8 | 9 | 10 | 11 |
| 6 | 7 | 8 | | | | | | | | |
| 9 | 10 | 11 | | | | | | | | |
| sr | | | | | | | | | | |

(rc x) perim t

| | | | | | | | | | | | | | | |
|----------------------|-----|--|---|---|---|---|---|---|---|---|---|---|----|----|
| 0 1 2 3 4 5 | abc | 0 1 2 3 | | | | | | | | | | | | |
| <div>us sr</div> | 12 | <table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr></table> <table><tr><td>6</td><td>7</td><td>8</td></tr><tr><td>9</td><td>10</td><td>11</td></tr></table> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 0 | 1 | 2 | | | | | | | | | | | | |
| 3 | 4 | 5 | | | | | | | | | | | | |
| 6 | 7 | 8 | | | | | | | | | | | | |
| 9 | 10 | 11 | | | | | | | | | | | | |

6.3 Formatted Display

x:"**y** is a literal representation of **y** specified by **x**. Positive elements of **x** specify fixed-point notation, while negative elements specify exponential notation. The left and right ranks are one; that is, lists in the arguments are independently formatted. The computation can be modelled as follows:

```
fmtexp    =. {&'++-'@* , _3&{.0('00'&,)@":@|
cexp      =. >:@(i.&'e') ({. , fmtexp@".@).) ]
cminus    =. '-'&{(e.&'_' # i.@#)@}})
larg      =. (+_20&*@(0&=))@-@{1&|}@|@".@(-.&' %e')
nsprintf  =. larg@[ cexp@cminus@": ]
psprintf  =. ".@(-.&' %f')@[ ($&' '0(0&=)@<.@[ , cminus@":) ]
sprintf   =. nsprintf`psprintf@.('f'&e.@{)

wd         =. <.@|
npstr      =. ' %- ' &,@(, &'e')@{0.1&":)@(-1&<)@|
ppstr      =. *@wd ). ' %&,@(, &'f')@{0.1&":)
pstr       =. npstr`ppstr@.(0&<:)

jexp       =. >:@(i.&'e') ({. , ":@".@(-.&' +' )@).) ]
jminus     =. ' _' &{(e.&'-' # i.@#)@}})
stars      =. }`{.@.( *@[] )` ($&' *'@[] )@.( *@[* .(<#) )
c2j        =. stars ]`jexp@.('e'&e.)@jminus

lb         =. {0&=@wd *. 0&<:)@{.
thcell     =. {wd@[ <@c2j pstr@[ sprintf ]]"0
thorn2     =. {lb@[ ]. ;@:thcell) " 1
```

The model is divided into groups of verbs.

sprintf is a limited model of **sprintf** in the C library, applying to a string containing a single **%e** or **%f** conversion specification and to a single number. Thus, if **embrace** =. ('{'&,)@(&'')', then:

```

embrace ' %0.3f' sprintf ^5      { 148.13}
embrace '%9.3f' sprintf ^_5      {   0.007}
embrace ' %- 0.3e' sprintf ^_5    {  6.738e-003}
embrace ' %- 9.3e' sprintf ^5     { -1.484e+002}
embrace ' %- 6.3e' sprintf ^_5    { -6.738e-003}

```

pstr applies to the left argument of **:** and produces the necessary left argument to **sprintf**. For example:

| x | embrace pstr x |
|-------------|-----------------------|
| _12 | { %- 11.0e} |
| _7.3 | { %- 6.3e} |
| _0.3 | { %- 0.3e} |
| 0 | { %0.0f} |
| 0.3 | { %0.3f} |
| 7.3 | {%7.3f} |
| 12 | {%12.0f} |

c2j and its constituents transform the result of **sprintf** to follow J conventions, in the treatment of negative signs (**jminus**), exponential notation (**jexp**), and overflow (**stars**).

thorn2 is a model of the dyad **:".** It works by applying **thcell** to corresponding atoms of the arguments, producing a list of boxes; the leading blank of the razed result is then dropped or not, according to the value of **lb** on left argument.

7. Comparatives

Comparisons between finite numbers are *tolerant*, as defined in Bernecky [1977]:

$$x = y \quad \text{if} \quad (|x-y| < :! .0 \text{ qct} * (|x| > . (|y|$$

(<:!.0 means *exact* less than or equal.) That is, x and y are tolerantly equal if the smaller is on or within the circle centered at the larger, having radius qct times the magnitude of the larger. qct , comparison tolerance, is a real number between 0 and $2^{_34}$ with a default value of $2^{_44}$; a non-default tolerance may be specified using the *fit* conjunction (!.).

Tolerant relations can be modelled as follows:

| | | |
|----------------------|--|--------------|
| <code>teq</code> | <code>=. @- <:!.0 qcts*0>.s </code> | in file ut.c |
| <code>tlt</code> | <code>=. < !.0 *. ~;</code> | ut.c |
| <code>tle</code> | <code>=. <:!.0 +. =</code> | ut.c |
| <code>tfloor</code> | <code>=. <:!.0 ([- -.@tle)]</code> | ut.c |
| <code>tceil</code> | <code>=. <:!.0 ([+ tlt)]</code> | ut.c |
| <code>dsignum</code> | <code>=. qct&<@ * 0&< - 0&></code> | ve.c |
| <code>jsignum</code> | <code>=. qct&<@ * (%)</code> | ve.c |

`teq`, `tlt`, and `tle` model tolerant equal, less than, and less than or equal. `tfloor` and `tceil` model tolerant floor and ceiling. `dsignum` computes the tolerant signum of a real number; `jsignum` that of a complex number.

Additionally, some comparisons internal to the system are *fuzzy*. Fuzzy comparisons are like tolerant comparisons, but depend on the parameter `qfuzz`, having fixed value $2^{_44}$. Such comparisons are used to decide whether arguments are in the domain of certain verbs; for example, `(2 3 +1e_14)$'abc'` is valid but `(2 3+1e_12)$'abc'` is not. Fuzzy comparisons can be modelled as follows:

```

int      =. (-2^31)&<: *. <&(2^31)
real     =. {.@+. "0

feq      =. |@- <:!.0 qfuzz&*@>.&|           in file ut.c
freal    =. >:!.0/@((qfuzz,1)&*)/@|@+.       ut.c

BfromD   =. ]` (1&=)@>(feq 1&=)              k.c
IfromD   =. ]`<.@.(int *. (feq<..))          k.c
DfromZ   =. ]`real @. (feq real)             k.c

```

The utility `int` tests for membership in the interval -2^{31} to $2^{31}-1$ inclusive. `real` produces the real part of a complex number. `feq` is 1 if its real arguments are equal within fuzz; `freal` is 1 if its complex argument is within fuzz of real. `BfromD`, `IfromD`, and `DfromZ` convert between types: boolean from real ("double"), integer from real, and real from complex.

8. Primitives

This chapter describes the primitives. Each entry has the spelling, class, program file, C object name, and notes on the implementation. The notes are mostly in the form of models written in J; the models are not necessarily optimal but are presented here because they are close to the implementation. The entries are ordered as in the dictionary, an order also shown in Appendix F on the back cover.

The following conventions and definitions apply:

| | |
|----------------|---|
| adv | Adverb |
| conj | Conjunction |
| m | Left noun argument to an adverb or a conjunction |
| n | Right noun argument to a conjunction |
| u | Left verb argument to an adverb or a conjunction |
| v | Right verb argument to a conjunction |
| pi | π , 3.14159265358979... |
| qct | Comparison tolerance (default: 2^{-44}) |
| qrl | Random link (initial value: 7^{-5}) |
| rk | An adverb that produces the ranks of its verb argument. For example, <code>*.rk</code> is <code>2 _ 2</code> |
| rank | <code>=. #0\$</code> Rank |
| mt | <code>=. 0&e.0\$</code> 1 if empty |
| id | <code>=. [.+</code> The identity adverb |
| type | <code>=. 3!:0</code> Type |
| complex | <code>=. 16&=0type</code> 1 if complex |
| boxed | <code>=. 32&=0type</code> 1 if boxed |
| pind | <code>=.]`]'`+0.(*0])"0</code> <code>n pind i</code> are integers in <code>i.n</code> |
| pfill | <code>=. [((i.0[-.]) ,]) pind</code> <code>n pfill p</code> converts <code>p</code> to a permutation of order <code>n</code> in the standard form; <i>i.e.</i> <code>(i.n) -: /:- n pfill p</code> |

```

= monad    vb.c    sclass    =. (((i.@#)=])#]) =/ ]>@.@(i.~)

= dyad      vb.c    eq

For atoms x and y, x=y is 1 if x equals y; tolerant equality
|@- <:!.0 qct&*@>.&| is used for numeric arguments. See 7
Comparatives and 3.3 Atomic Verbs.

=.          p.c      is1          See 1.4 Name Resolution.

=:          p.c      isg          See 1.4 Name Resolution.

< monad     v.c      box

<y has the following properties:
0 = rank <y          atomic
y -: >y             open is the inverse of box
y ~: <y             box y differs from y
32 = type <y         the type is encoded as 32

< dyad      vb.c      lt          =. <:!.0 *. ~:

<. monad     ve.c      floor1

floor =. <:!.0      NB. a function in the C library
dfloor =. {} - <) floor@(.0.5&+)
zfl =. floor@+.
inc =. (1&<:@(+)) * 1 0&=@(>:!.0/)) @ (+. - zfl)
zfloor =. zfl j./@:+ inc
floor1 =. dfloor`zfloor @. complex " 0 " _

See also 3.3 Atomic Verbs.

<. dyad      ve.c      minimum    See 3.3 Atomic Verbs.

<: monad     ve.c      decrem      =. -&1

<: dyad      vb.c      le          =. <:!.0 +. =

```

```

> monad    v.c    ope

mrk      =. >./@:(rank@>)
crk      =. mrk (-@[{.$&1@[, $@])&.> ]
crank    =. crk ($,&.> ]
msh      =. >./@:($@>)
cshape   =. <@msh {,&.> ]
mtp      =. >./@:((type*-.@mt)@>)
fill     =. >@({&(' ';(<$0);0))@ (2 32&i.)
ctype    =. (msh <@$ fill@mtp) (]'[@.(mt@)])&.> ]
ope      =. > @ cshape @ ctype @ crank

```

See Section II.B of the dictionary.

```

> dyad      vb.c    gt          =. -.@<:

>. monad    ve.c    ceil1       =. <,&.-

>. dyad      ve.c    maximum     =. <,&.-"0

>: monad     ve.c    increm      =. 1&+

>: dyad      vb.c    ge          =. -.@<

_ noun       w.c     coninf       See 1.1 Word Formation.

_ noun       w.c     coninf       See 1.1 Word Formation.

_: monad     v.c     inf1         =. _" _

_: dyad      v.c     inf2         =. _" _

+ monad      ve.c     conjug      See 3.3 Atomic Verbs.

+ dyad       ve.c     plus        See 3.3 Atomic Verbs.

```

| | | | |
|------------------------|-------------------|---------------------|---|
| <code>+. monad</code> | <code>vm.c</code> | <code>rect</code> | <code>=. 9 11&o."0" _</code> |
| <code>+. dyad</code> | <code>ve.c</code> | <code>gcd</code> | See 3.3 <i>Atomic Verbs</i> . |
| <code>+: monad</code> | <code>ve.c</code> | <code>duble</code> | <code>=. +~</code> |
| <code>+: dyad</code> | <code>vb.c</code> | <code>nor</code> | <code>=. -.@+.</code> |
| <code>* monad</code> | <code>ve.c</code> | <code>signum</code> | <code>=. (%) * >!.0&qct@ </code> |
| <code>* dyad</code> | <code>ve.c</code> | <code>tymes</code> | See 3.3 <i>Atomic Verbs</i> . |
| <code>*. monad</code> | <code>vm.c</code> | <code>polar</code> | <code>=. 10 12&o."0" _</code> |
| <code>*. dyad</code> | <code>ve.c</code> | <code>lcm</code> | See 3.3 <i>Atomic Verbs</i> . |
| <code>*: monad</code> | <code>ve.c</code> | <code>square</code> | <code>=. *~</code> |
| <code>*: dyad</code> | <code>vb.c</code> | <code>nand</code> | <code>=. -.@*.</code> |
| <code>- monad</code> | <code>ve.c</code> | <code>negate</code> | <code>=. 0&-</code> |
| <code>- dyad</code> | <code>ve.c</code> | <code>minus</code> | See 3.3 <i>Atomic Verbs</i> . |
| <code>-. monad</code> | <code>ve.c</code> | <code>not</code> | <code>=. 1&-</code> |
| <code>-. dyad</code> | <code>v.c</code> | <code>less</code> | <pre> dr =. rank@] - 0&>.@<:@rank@[res =. (dr (*/@{. ,).) \$@) \$,@] less =. [`(((-.@e. res) # [])@.((<: >:)&rank) </pre> |
| <code>-.: monad</code> | <code>ve.c</code> | <code>halve</code> | <code>=. %&2</code> |


```
-: dyad      vb.c  match
```

```

x -: y if
-: &(#@,)      numbers of elements match; and
-: &rank        ranks match; and
-: &$           shapes match; and
*./@(&,&)      corresponding atoms match

```

```
% monad     ve.c  recip      =. 1&%
```

```
% dyad      ve.c  divide     See 3.3 Atomic Verbs.
```

```
% monad     vi.c  minv
```

minv has two main constituents: **qr** computes the QR decomposition; **rinv** computes the inverse of a square upper triangular matrix.

```

pdt  =. +/ . *
en   =. 1&{ @ (, &1 1) @ $

t=.0 0$' '
t=.t, 'n  =. en y.'
t=.t, 'm  =. >.-: n'
t=.t, 'a0 =. m{."1 y.'
t=.t, 'a1 =. m}."1 y.'
t=.t, 't0 =. qr a0'
t=.t, 'q0 =. >@{. t0'
t=.t, 'r0 =. >@{: t0'
t=.t, 'c  =. (+|:q0) pdt a1'
t=.t, 't1 =. qr a1 - q0 pdt c'
t=.t, 'q1 =. >@{. t1'
t=.t, 'r1 =. >@{: t1'
t=.t, '(q0,q1);(r0,c),(-n){."1 r1'
q2   =. t : ' '
norm =. (%:pdt +)@,
qr    =. q2`(({% ;&, ,~@en@[ $ ]) norm) @. (1&>:@en)

```

```

x=.0 0$''
x=.x, 'n =. #y.'
x=.x, 'm =. >.-: n'
x=.x, 'ai =. rinv (m,m){.y.'
x=.x, 'di =. rinv (m,m){.y.'
x=.x, 'b =. (m,m-n){.y.'
x=.x, 'bx =. - ai pdt b pdt di'
x=.x, '(ai, bx), (-n){."1 di'
r4 =. x : ''
rinv =. r4`% @. (1%>:@#)

minv =. (|.@$ ($,) (rinv@] pdt +@|:@()&>/@qr) " 2

%: dyad      vi.c   mdiv      =. (%@] +/ . * []) " _ 2

%: monad     vm.c   sqrt      =. 2%:

%: dyad      vm.c   root      =. (] ^ %@]"0

^ monad      vm.c   expn1

exp =. ^      NB. a function in the C library
sin =. 1%o.
cos =. 2%o.
zexp =. ((^@[ * cos@])j. (^@[ * sin@]))/@+.
expn1 =. exp`zexp @. complex " 0 " _

^ dyad      vm.c   expn2      =. ^@(^.[ * ])"0

^: monad     vm.c   logar1

atan2 =. 12%o.@j. NB. a function in the C library
logar1 =. (^.[| j. atan2/@+.)"0"_

^: dyad      vm.c   logar2      =. %~%^. "0

^: conj.     cp.c   powop

```

\$ monad vs.c **shape** See 2.1 *Arrays*.

\$ dyad vs.c **reitem** =. ((,).@\$(\$,)])"1 _

\$. cx.c **ensuite**

\$. monad p.c **self1**

\$. dyad p.c **self2**

~ adverb a.c **swap**

m~ is a reference to the verb named by **m**. See 1.4 *Name Resolution*.

u~ is (] u [])"(_ 2 1(u rk)).

~. monad v.c **nub** =. ~: #]

~: monad vb.c **nubsieve** =. i.@# = i.~

~: dyad vb.c **ne** =. -.@=

| monad ve.c **mag** =. (>.)`(%:@*+)@.complex

| dyad ve.c **residue** See 3.3 *Atomic Verbs*.

|. monad vs.c **reverse** =. {- i.@-@#

|. dyad vs.c **rotate**

rotate =.]`(((i.@]-]-|~)#){])@.(*@rank@)"0 _

|: monad vs.c **cant1** =. i.@-@rank |:]

|: dyad vs.c **cant2**

mask =. =/ i.@>:@(>./)

vec =. >@{@:(i.&.>)@((<./ .+) _&*@-.)

ind =. vec +/ .* (#. |:)

```

canta =. ($@] ind mask@] { ,@]

en      =. - #@;
ci      =. (:@pfill ;) { i.@en , en + (#@> # i.@#)@]
cant2 =. ((rank@] ci []) canta ]) " 1 _

```

See Hui [1987] 3.1. `canta` is dyadic transpose in APL.

```

. conj.      c.c      dot

minors =. (0 0 1&.) @ (1&([\.])
col     =. (:@ (1&,)@)
monad   v/@,` (u@,) ` ({"1 u . v$:@minors)@.(0 1&i.@col)"2
dyad    x u . v y is
        x u@ (v" (lv,lv>.<: # $y) " (1+lv,_) ) y [ lv=.1{v rk

```

See Hui [1987] 3.3.

```

.. conj      c.c      even      =. [.` (-: @: + [.] ) `& \

```

```

.: conj      c.c      odd       =. [.` (-: @: - [.] ) `& \

```

```

: conj.      cx.c      colon

```

```

.: conj.      c.c      obverse   See 3.4 Obverses.

```

```

, monad      vs.c      ravel

```

`,y` has the following properties:

```

1          -: rank ,y
(*/$y) -: # ,y
y          -: ($y)$ ,y

```

```

, dyad       vs.c      over

```

```

.. monad     vs.c      table      =. {#, */@}.@ $ ,

```

```

.. dyad      vs.c      overrr     =. ,"_1

```

```
;: monad    vs.c    lamin1      =. 1&,@$ $ ,
```

```
;: dyad      vs.c    lamin2
```

```
v00 =.      [ , ,.@]
```

```
v01 =.      [ , ,:@]
```

```
v10 =. ,:@[ ,      ]
```

```
v11 =. ,:@[ , ,:@]
```

```
lamin2 =. v00`v01`v10`v11 @. (#.@*@,&rank)
```

```
; monad      v.c      raze
```

The monad `;` is `>@(&.>/)@, .` This is an $O(n^2)$ algorithm. The implementation uses a faster method — copying items from the argument into a pre-allocated space — when `1&>:@#@~.@:(type@>)` and `1&>:@(>./ - <./)@:(rank@>)`.

```
; dyad        v.c      link      =. <@[ , <`]@.boxed@]
```

```
;: conj.      cc.c      cut
```

```
cut_1 =. (&.) (: .1)
```

```
cut2  =. (&|. ) (: .1) (&|. )
```

```
cut_2 =. (&): (: .2)
```

See Hui [1987] 3.2.

```
;: monad      w.c      words      See 1.1 Word Formation.
```

```
# monad      vs.c      tally      =. {.@(&1)@$
```

```
# dyad        vs.c      repeat     =. ;@(<@($,:)"_1) " 1 _
```

```
#. monad      vc.c      base1      =. 2&#."1
```

```

#. dyad      vc.c  base2

ext  =. (#@) # []`[ @. (*@rank@)
base2 =. (* /\ .@).@ (, &1)@ext +/ . * ] " 1

#: monad     vc.c  abase1

max   =. >./@|@,
bits  =. >:@<.@(2&^.)@ (1&>.)
abase1 =. #:~ $&2@bits@max

#: dyad      vc.c  abase2      =. ([ | ((%~)-|) /\ .@).@,) "1 0

! monad      vm.c  fact

! dyad       vm.c  outof

case =. #. @ (0&>*. (= <.) ) @ ([ , , --)
f000 =. !@] % !@[ * !@--
f001 =. 0:
f010 =. 'domain error' "0
f011 =. _1&^@[ * [ ! (->:)
f100 =. 0:
f101 =. 'can not happen' "0
f110 =. _1&^@-- * !&|&>::~
f111 =. 0:
outof =. f000`f001`f010`f011`f100`f101`f110`f111 @.case"0

```

See *SHARP APL Reference Manual*, pp. 131-133 (Berry [1979]) and 3.3 *Atomic Verbs*.

```

!. conj.      cf.c  fit          See 3.4 Variants.

!: conj.      x.c   foreign

```

The `!` conjunction takes integer scalar left and right arguments, and produces verbs. (One exception: `5! : 0` is an adverb.) These verbs behave like other verbs; in particular, they have intrinsic ranks, may be assigned names, and may serve as arguments to adverbs and

conjunctions. Where these verbs take names as arguments (file names, workspace names, or object names), the names are always boxed, and the verb rank is 0.

See Appendix E for the names of functions which implement the various cases of `m!:n`.

```
/ adverb   a.c   slash
```

```
/. adverb  ap.c  sldot
```

```
key      =. (@#) (=@[`) (`]) \
osub     =. >@]`(>@[ >@:( ] ) @. (*@#@])
oind     =. (+/&i./ </.&, i.)@{2&{.)@{(&1 1)@$
oblique  =. (@(osub"0 1)) (oind`) (`(&(<"_2))) \
sldot    =. id (oblique : key)
```

```
/: monad   vg.c   gradel
```

```
qsort     NB. a function in the C library
arg       =. <"_1 ,. ]&.>@i.@#
gradel    =. >@{:"1 @ qsort @ arg
```

```
/: dyad     vg.c   grade2      =. {~ /:
```

```
\ adverb   ap.c   bslash
```

```
base      =. 1&>.@-@[ * i.@em
iind       =. base ,. |@[ <. en - base
seg        =. ((+i.)/[ { ]) "1 _
infix      =. (@seg) (iind`) (`]) \ ("0 _)
prefix     =. (@{.) (>:@. @i.@#`) (`]) \
bslash     =. id (prefix : infix)
```

```
\. adverb  ap.c   bsdot
```

```
en        =. #@]
em         =. (en >. @% 1&>. @|@[])`(en 0&>. @>:@- [] ) @. (0&<:@[])
```

```

kay    =. en`em @. (0&<@[])
omask  =. (em,en) $ ($&0@|@[ , $&1@kay)
outfix =. (@#) (omask`) (`]) \ ("0 _)
suffix =. (@).) ~ (`(, .@i.@#)) \
bsdots =. id (suffix : outfix)

```

```

\: monad    vg.c    dgrade1

```

```

qsort      NB. a function in the C library
darg       =. <"_1 ,. -&.>@i.@#
dgrade1    =. |.@- @: (>@("1) @ qsort @ darg

```

```

\: dyad      vg.c    dgrade2    =. {~ \:

```

```

[ monad      v.c    left1

```

```

[ dyad       v.c    left2

```

```

[. conj.     c.c    lev

```

```

] monad      v.c    right1

```

```

] dyad       v.c    right2

```

```

]. conj.     c.c    dex

```

```

{ monad      vs.c    catalog

```

```

count      =. */@>
prod       =. */\.@).@(&1)
copy       =. */@[ $> prod@[ (#,)&.> ]
catalog    =. (:@:($&.>) $ count <"1@|:@copy ]) " 1

```

See Hui [1987] 2.1.


```
{ dyad      vs.c      from

ifrom =. (#@) pind [] >@{ <"_1@}
afi   =. pind` (i.@[-.(pind>)) @. (boxed@)
afrom =. ($@) #. $@ >@(@:(afi&.) >@() ifrom ,@)
from  =. ifrom`afrom @. (boxed@) " 0 _
```

See Hui [1987] 2.2.

```
{. monad    vs.c      head      =. 0@{

{. dyad      vs.c      take

fill  =. >@({&(' '(<$0);0)) @ (2 32&i.@(type*-.@mt))
pad   =. fill@ $~ (|@[-#@) 0) $@
ti    =. i.@-@[ + [ + #@
case  =. 0@<:@[ #.@, |@[ > #@
itake =. (ti[])`([],~pad)`(i.@[[])`([],pad) @. case
taker =. '':'({.x.) itake"({:x.) y.'
raise =. (1"0@[ $ ])`@. (*@rank@)
larg  =. <@,"(0) _&(0))@-@i.@#
targ  =. larg@[ , <@raise
take  =. >@ (taker&./)@targ " 1 _
```

```
{: monad    vs.c      tail      =. _1@{
```

```
} adverb    a.c      rbrace

m)          m"_]
monad u)    , {~ i.@).@ $ + */@).@ $ * # pind u
dyad u)     (i.@$@) (|. @, &, i.[] pind@u) { ] |. @, &, $@u $ [
```

See Hui [1987] 2.4.

```
} monad     vs.c      behead     =. 1@{.
```

```

}. dyad      vs.c   drop

pi   =. 0&<@[ * 0&<.@-
ni   =. 0&>:[ * 0&>.@+
di   =. ({.~ rank) (pi + ni) $@]
drop =. (di {. 1)"1 _

}: monad     vs.c   curtail      =. _1&}.

" conj       cr.c   qq           See 3.2 Rank.

". monad     v.c    exec1

". dyad      v.c    exec2

": monad     f.c    thorn1       See 6 Display.

": dyad      f.c    thorn2       See 6.3 Formatted Display.

^ conj.      cg.c   tie

ar          an adverb that produces atomic representation of a verb
m`n        m,n
m`v        m,(v ar)
u`n        (u ar),n
u`v        (u ar),(v ar)

^: conj.     cg.c   evger

@ conj.      c.c    atop

@. conj.     c.c    agenda

For argument cells x and y of m@.v:
m@.v y      is ((v y){m})` : 0 y
x m@.v y    is x ((x v y){m})` : 0 y

```

@: conj c.c atco =. [.@("_)

& conj. c.c amp

m&v Empty dyadic domains; infinite monadic rank.

u&n Empty dyadic domains; infinite monadic rank.

u&v u@v : (v@[u v@]) " ({.v rk)

&. conj. c.c under =. (.)^(^_1))@&

&: conj. c.c ampco =. [.&("_)

? monad v.c roll

tick =. [<.@%~ (* 'qrl=: (<:2^31) | (7^5)*qrl': '')@]

roll =. (<:2^31)&tick"0

See *SHARP APL Reference Manual*, p. 126 (Berry [1979]).

? dyad v.c deal

tick =. [<.@%~ (* 'qrl=: (<:2^31) | (7^5)*qrl': '')@]

step =. <@~.@((+ (2^31)&tick)/\)@[C.]

arg =. <@i.@-@] ,~ i.@-@[([,&.> -~)]

deal =. ([(. >@ (step&.>/)@arg)"0

See *SHARP APL Reference Manual*, p. 178 (Berry [1979]).

) cx.c label

a. noun j.c alp The 256-letter ASCII alphabet.

A. monad vp.c adot1

ord =. >:@(>./)

base =. >:@i.@-@#

rfd =. +/@({.>).)\.

dfr =. /: ^:2@,/

adot1 =. (base #. rfd)@((ord pfill])`C.@.boxed) " 1

```

A. dyad      vp.c  adot2      =. dfr@ (base@) #: [] { }

b. adverb    a.c   bool

tt  =. i.@rank |: {&(#:i.16)
bool =. '0&$: : (+:0[ {&(tt x.)@+ ])' " _ 0 0' : 1

c. monad     vm.c  eig1      Not yet available

c. dyad      vm.c  eig2      Not yet available

c. monad     vp.c  cdot1

ac   =. (, i. ]) { 1&|.@[ , ]
dfc  =. >@ (ac&.>/)@ (pind&.> , <@i.@[)
bc   =. <@ (([ i. >./) |. ])@~.
cfd  =. ~.@ (/ : {.&>)@: (bc"1)@|: @ ({/\)@ (, ~@[ $ pfill)
cdot1 =. (ord cfd )` (ord@; dfc )@.boxed " 1

c. dyad      vp.c  cdot2

cdot2 =. ((#@ pfill`dfc@.(boxed@) []) { }) " 1 _

e. monad     vb.c  razein     =. e.~&> <@;

e. dyad      vb.c  eps        =. i.~ < #@]

E. dyad      vb.c  ebar      Not yet available

f. adverb    a.c   fix

i. monad     v.c   iota

rev  =. '' : '|."x. y.'
ineg =. # - 0&> # i.@#
iota =. > @ (rev&.>/) @ (<"0@ineg , (<@ $ i.@(*))@|) " 1

```

i. dyad vm.c indexof

If **x** and **y** are literal lists, then **x i. y** is **x ciof y**:

```
map =. '(i.-#y.) (a.i.|.y.))256$#y.' ; ''
ciof =. a.&i.0] { map@[
```

Otherwise, if **x** and **y** are not floating point or complex numbers, or if the comparison tolerance **qct** is zero, a straightforward hashed algorithm is used.

Otherwise, if **x** or **y** are floating point numbers and **qct** is nonzero, an algorithm due to Arthur Whitney is used:

bit x Convert a floating point **x** into a Boolean vector
tib b Convert a Boolean vector **b** into a floating point number
hash b Hash function on a Boolean vector **b**

There exists a Boolean **mask** with a minimum number of ones such that **tib mask*.bit x** is within **qct** of **x**; the actual mask used in the algorithm may have fewer number of ones. For each **xi** of **x**, compute **hash mask*.bit xi**; for each **yj** of **y**, compute:

```
hl =. hash mask*.bit yj*1-qct
hr =. hash mask*.bit yj*1+qct
```

Look for **hl** and **hr** in the list of hashed **xi**'s. In other words, if **hash** were a perfect hash, then for **th=.hash@(mask&*.)@ bit"0**, **x i. y** is **((th x)i.th y*1-qct)<.(th x)i.th y*1+qct)**.

j. monad vm.c jdot1 =. 0j1&*

j. dyad vm.c jdot2 =. (+ j.)"0

NB. w.c wordi1 See 1.1 *Word Formation*.

o. monad vm.c pix =. pi&*

o. dyad vm.c circle

```

sin      =. 1&o.          NB. a function in the C library
cos      =. 2&o.          NB. a function in the C library
sinh     =. 5&o.          NB. a function in the C library
cosh     =. 6&o.          NB. a function in the C library

cir0     =. 1&+    %:@* 1&-
zp4      =. -&0j1 %:@* +&0j1
zp8      =. 0j1&+ %:@* 0j1&-
zm4      =. +&1 * -&1 %:@% +&1
real     =. -:@(++ )
imag     =. %&0j2@(-+)
zarc     =. 0j_1&*@^.@*^0: @. (0&=)

zsin     =. ((sin@[ * cosh@]) j. ( cos@[ * sinh@]))/@+.
zcos     =. ((cos@[ * cosh@]) j. (-@sin@[ * sinh@]))/@+.
ztan     =. zsin % zcos
zsinh    =. zsin&.j.
zcosh    =. zcos@j.
ztanh    =. ztan&.j.

zasin    =. zsinh&.j.
zacos    =. (-:pi)&-@zasin
zatan    =. zatanh&.j.
zasinh   =. (^.@+ zp4)^(%&.-) @. (0&>@real)
zacosh   =. ]^(j.@|@imag)@.(0&>@real) @ (^.@+ zm4)
zatanh   =. 1&+ -:@^.@% 1&-

cirp     =. (cir0@)]^(zsin@)]^(zcos@)]^(ztan@)]^(zp4@)]^(
           (zsinh@)]^(zcosh@)]^(ztanh@)]^(zp8@)]^(
           (real@)]^(|@)]^(imag@)]^(zarc@)] @. [
cirm     =. (cir0@)]^(zasin@)]^(zacos@)]^(zatan@)]^(zm4@)]^(
           (zasinh@)]^(zacosh@)]^(zatanh@)]^(~@zp8@)]^(
           ]^(+@)]^(j.@)]^(r.@)] @. (|@)]
circle   =. cirp`cirm @. (0&>@[] " 0

```

See *Handbook of Mathematical Functions*, Chapter 4 (Abramowitz and Stegun [1964]).

| | | | |
|-----------------|------|--------------|--------------------|
| p. monad | vm.c | poly1 | Not yet available. |
| p. dyad | vm.c | poly2 | Not yet available. |
| r. monad | vm.c | rdot1 | =. ^@j. |
| r. dyad | vm.c | rdot2 | =. (* r.)"0 |
| x. | CX.C | xd | |
| y. | CX.C | xd | |
| 0: monad | v.c | zero1 | =. 0"_ |
| 0: dyad | v.c | zero2 | =. 0"_ |
| 1: monad | v.c | one1 | =. 1"_ |
| 1: dyad | v.c | one2 | =. 1"_ |

Appendix A. Incunabulum

One summer weekend in 1989, Arthur Whitney visited Ken Iverson at Kiln farm and produced — on one page and in one afternoon — an interpreter fragment on the AT&T 3B1 computer. I studied this interpreter for about a week for its organization and programming style; and on Sunday, August 27, 1989, at about four o'clock in the afternoon, wrote the first line of code that became the implementation described in this book.

Arthur's one-page interpreter fragment is as follows:

```
typedef char C;typedef long I;
typedef struct a(I t,r,d[3],p[2];)*A;
#define P printf
#define R return
#define V1(f) A f(w)A w;
#define V2(f) A f(a,w)A a,w;
#define DO(n,x) {I i=0,_n=(n);for(;i<_n;++i){x;}}
I *ma(n){R(I*)malloc(n*4);}mv(d,a,n)I *d,*a;{DO(n,d[i]=a[i]);}
tr(r,d)I *d;{I z=1;DO(r,z=z*d[i]);R z;}
A ga(t,r,d)I *d;{A z=(A)ma(5+tr(r,d));z->t=t,z->r=r,mv(z->d,d,r);
R z;}
V1(iota){I n=w->p;A z=ga(0,1,&n);DO(n,z->p[i]=i);R z;}
V2(plus){I r=w->r,*d=w->d,n=tr(r,d);A z=ga(0,r,d);
DO(n,z->p[i]=a->p[i]+w->p[i]);R z;}
V2(from){I r=w->r-1,*d=w->d+1,n=tr(r,d);
A z=ga(w->t,r,d);mv(z->p,w->p+(n**a->p),n);R z;}
V1(box){A z=ga(1,0,0);*z->p=(I)w;R z;}
V2(cat){I an=tr(a->r,a->d),wn=tr(w->r,w->d),n=an+wn;
A z=ga(w->t,1,&n);mv(z->p,a->p,an);mv(z->p+an,w->p,wn);R z;}
V2(find){}
V2(rsh){I r=a->r?*a->d:1,n=tr(r,a->p),wn=tr(w->r,w->d);
A z=ga(w->t,r,a->p);mv(z->p,w->p,wn=n>wn?wn:n);
if(n==wn)mv(z->p+wn,z->p,n);R z;}
V1(sha){A z=ga(0,1,&w->r);mv(z->p,w->d,w->r);R z;}
V1(id){R w;}V1(size){A z=ga(0,0,0);*z->p=w->r?*w->d:1;R z;}
pi(i){P("%d ",i);nl(){P("\n");}}
```

```

pr(w)A w:{I r=w->r,*d=w->d,n=tr(r,d);DO(r,pi(d[i]));nl();
  if(w->t)DO(n,P("< " );pr(w->p[i]))else DO(n,pi(w->p[i]));nl();}

C vt[]="+{~<#,";
A(*vd[])()={0,plus,from,find,0,resh,cat},
(*vm[])()={0,id,size,iota,box,sha,0};
I st[26]; qp(a){R a>='a'&&a<='z';}qv(a){R a<'a';}
A ex(a)I *a;{I a=*a;
  if(qp(a)){if(e[1]=='')R st[a-'a']=ex(e+2);e= st[ a-'a'];}
  R qv(a)?(*vm[a])(ex(e+1)):e[1]?(*vd[e[1]])(a,ex(e+2)):(A)a;}
noun(c){A z;if(c<'0'||c>'9')R 0;zaqa(0,0,0);*z->p=c-'0';R z;}
verb(c){I i=0;for(;vt[i];)if(vt[i++]==c)R i;R 0;}
I *wd(s)C *a;{I a,n=strlen(s),*e=ma(n+1);C c;
  DO(n,e[i]=(a=noun(c=s[i]))?a:(a=verb(c))?a:c);e[n]=0;R e;}

main(){C s[99];while(gets(s))pr(ex(wd(s)));}

```

Appendix B. Program Files

| | |
|------|---|
| a.c | adverbs |
| ai.c | adverbs — inverse and identity |
| ap.c | adverbs — partitions |
| au.c | adverbs — utilities |
| c.c | conjunctions |
| cc.c | conjunctions — cuts |
| cf.c | conjunctions — fit |
| cg.c | conjunctions — gerunds |
| cp.c | conjunctions — power |
| cr.c | conjunctions — rank |
| ct.c | conjunctions — trains |
| cx.c | conjunctions — explicit definition |
| f.c | format (display) |
| i.c | initialization |
| io.c | input/output |
| j.c | main and global variables |
| k.c | conversion |
| m.c | memory management |
| p.c | parsing |
| pc.c | parsing — tacit conjunction translator |
| pv.c | parsing — tacit verb translator |
| r.c | representation |
| rt.c | tree representation |
| s.c | symbol table |
| t.c | tables |
| u.c | utilities |
| ut.c | utilities — tolerant and fuzzy comparison |
| v.c | verbs |
| vb.c | verbs — boolean |
| ve.c | verbs — elementary functions |

| | |
|--------|--|
| vg.c | verbs — grades |
| vh.c | verbs — hashed indexing |
| vi.c | verbs — matrix inverse and matrix divide |
| vm.c | verbs — mathematical functions |
| vp.c | verbs — permutation |
| vs.c | verbs — selection & structural |
| vz.c | verbs — complex functions |
| w.c | word formation |
| x.c | external, experimental, and extra |
| xf.c | external — files |
| xs.c | external — scripts |
| xw.c | external — workspaces |
| a.h | adverbs and conjunctions |
| io.h | input/output |
| j.h | global definitions |
| jc.h | character definitions |
| je.h | extern declarations |
| jt.h | types |
| p.h | parsing |
| v.h | verbs |
| x.h | external, experimental, and extra |
| lj.c | LinkJ |
| lj.h | LinkJ |
| main.c | LinkJ example |

Appendix C. The LinkJ Interface

LinkJ is a set of object modules which together offer the full capability of J while allowing links to other compiled routines and libraries. It is possible to call J from C and to call C from J. The interface consists of the following definitions, functions, and variables:

```
typedef char B;
typedef char C;
typedef long I;
typedef struct{I t,c,n,r,s[1];}*A;
typedef A (*AF)();

C jinit(void);           B asgn;
A jx(C*s);              C jerr;
A jpr(A x);
A jma(I t,I n,I r);
C jfr(A x);
A jset(C*name,A x);
C jc(I k,AF*f1,AF*f2);
```

A is the C data type of an array. The parts are the type, reference count, number of elements in the array, rank, shape, and the array elements, in a contiguous segment of memory. (Array types are boolean, literal, integer, floating point, complex, and boxed. See file lj.h.) **AF** typifies a function which accepts one or more array arguments, and returns an array result; that is, **AF** is the C data type of a verb.

jinit initializes J. **jx** applies to a 0-terminated string representing a sentence, and returns the array result of executing the sentence; the global variable **asgn** is 1 if the last operation is assignment. When an error is encountered in an interface function, the result is 0, and the global variable **jerr** contains an error number as defined in file lj.h. For example:

```
jinit();
p=jx("a=.i.3 4");
q=jx("+/,a");
```

p is a 3 by 4 table of the integers from 0 to 11, and **q** is the atom 66. The space occupied by the result of **jx** is reused the next time **jx** is called.

jpr(x) prints array **x** on the standard output; the result is **x** itself.

jma(t,n,r) allocates memory for an array of type **t** having **n** elements and rank **r**. (The shape and the elements must then be filled.) **jfr** frees an array previously allocated by **jma**. Array arguments and results must use space managed by **jma** and **jfr**.

jset(name,x) assigns a value to a global name (as in the copula **=:**). **name** is a 0-terminated string; **x** is an array. The result of **jset** is **x** itself. **jx(name)** returns the referent of a name.

The preceding functions allow calling J from C. The following facilities allow calling C from J. A new case of the **!:** foreign conjunction is defined: **10!:k** is a verb whose definition is controlled by **jc**, a function written by the user, as follows:

```
C jc(I k,AF*f1,AF*f2) {
    switch(k) {
        /* k:   index                                */
        /* f1:  pointer to monad (or NULL if no monad) */
        /* f2:  pointer to dyad (or NULL if no dyad)  */
        /* result is 0 if there is an error, nonzero if no error */
    }
}
```

10!:k invokes **jc(k,&f1,&f2)**, wherein (presumably depending on **k**) ***f1** is assigned a pointer to a monadic function and ***f2** a pointer to a dyadic function. The result of **10!:k** is a verb like any other; in particular, it may be assigned a name and may serve as argument to adverbs and conjunction; and when it is invoked with arguments the functions assigned to ***f1** and ***f2** are invoked with those arguments.

File **main.c** contains an example of using LinkJ. It has a **main** function which repeats the following steps, *ad infinitum*:

- Get a line of input from the terminal;
- Execute the line;
- Print the error number if an error occurred, or the result if the last operation was not assignment.

(To terminate, enter CTRL D or execute `0!:55 ''`.) As well, `main.c` has an example of using `jc: 10!:0 y` computes `#,y`, the number of elements in array `y`; and `x 10!:0 y` computes `x{.,y`, the first `x` elements of integer array `y`.

Appendix D. Compiling

There is a single set of program files; machine and compiler dependencies are handled by conditional compilation (`#if` preprocessor statements). The following names must be defined in file `j.h`:

```
#define SYS                SYS_???

#define LINKJ              0
#define WATERLOO          0

#define SYS_ANSILIB       0
#define SYS_LILENDIAN     0
#define SYS_SESM          0
#define SYS_UNIX          0
```

`SYS` identifies the current system, and must be one of the `SYS_*` names defined at the beginning of `j.h` — `SYS_PCAT`, `SYS_MACINTOSH`, `SYS_SUN4`, etc. The inclusion of a system name in the list of `SYS_*` names does not imply that the program files would compile in that system, nor that the compiled result would work. (The file `status.doc` has a list of working systems.)

`LINKJ` and `WATERLOO` are Boolean flags. `LINKJ` is set to 1 to generate the LinkJ modules. (See Appendix C *The LinkJ interface*.) `WATERLOO` is set to 1 when compiling on machines at the University of Waterloo using the MFCF library organization.

The other `SYS_*` names are Boolean masks, used as `(SYS & SYS_UNIX)`. In compiling on a machine from the existing list, these masks can remain unchanged; otherwise, in porting to a new system, it is easiest just to set a mask to 0 or 1 as appropriate. `SYS_ANSILIB` selects systems using the ANSI C library organization. `SYS_LILENDIAN` selects “little endian” (reverse byte order) systems. The PC (Intel 80x86) line of machines are little endian. `SYS_SESM` selects systems using the J session manager. (The session manager is not publicly available, so `SYS_SESM` should be set to 0.) `SYS_UNIX` selects UNIX systems.

Object modules must be linked with the C math library to generate an executable module. The procedure varies from system to system; the command `cc *.o -lm -o j` works under UNIX.

Bibliography

- Abramowitz, M., and I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, 1964 6.
- Bernecky, R., *Comparison Tolerance*, SHARP APL Technical Notes 23, 1977 6 10.
- Bernecky, R., G. Bezoff, and M. Symes, *APL Workspace Transfer*, SHARP APL Technical Notes 22, Revision 4, 1981 7 15.
- Bernecky, R., and R.K.W. Hui, *Gerunds and Representations*, APL91, APL Quote-Quad, Volume 21, Number 4, 1991 8.
- Berry, P.C., *SHARP APL Reference Manual*, I.P. Sharp Associates, 1979 3; Additions and Corrections, 1981 6.
- Falkoff, A.D., and K.E. Iverson, *APL\360 User's Manual*, IBM Corporation, 1968 8.
- Falkoff, A.D., and K.E. Iverson, *The Design of APL*, IBM Journal of Research and Development, Volume 17, Number 4, 1973 7.
- Falkoff, A.D., and K.E. Iverson, *The Evolution of APL*, ACM SIGPLAN Notices, Volume 13, Number 8, 1978 8.
- Hui, R.K.W., *Some Uses of { and }*, APL87, APL Quote-Quad, Volume 17, Number 4, 1987 5.
- Hui, R.K.W., *An Implementation of J*, Iverson Software Inc., 1992 1 27.
- Hui, R.K.W., K.E. Iverson, and E.E. McDonnell, *Tacit Definition*, APL91, APL Quote-Quad, Volume 21, Number 4, 1991 8.
- Hui, R.K.W., K.E. Iverson, E.E. McDonnell, and A.T. Whitney, *APL\?*, APL90, APL Quote-Quad, Volume 20, Number 4, 1990 7.
- Iverson, K.E., *A Programming Language*, Wiley, 1962 5.
- Iverson, K.E., *Operators and Functions*, Research Report #RC7091, IBM Corporation, 1978 4 26.

- Iverson, K.E., *Notation as a Tool of Thought*, Communications of the ACM, Volume 23, Number 8, 1980 8.
- Iverson, K.E., *Rationalized APL*, I.P. Sharp Associates, 1983 1 6.
- Iverson, K.E., *APL87*, APL87, APL Quote-Quad, Volume 17, Number 4, 1987 5.
- Iverson, K.E., *A Dictionary of APL*, APL Quote-Quad, Volume 18, Number 1, 1987 9.
- Iverson, K.E., *Arithmetic*, Iverson Software Inc., 1991.
- Iverson, K.E., *ISI Dictionary of J*, Version 4, Iverson Software Inc., 1991.
- Iverson, K.E., *Programming in J*, Iverson Software Inc., 1991.
- Iverson, K.E., *Tangible Math*, Iverson Software Inc., 1991.
- Iverson, K.E., *A Personal View of APL*, IBM Systems Journal, Volume 30, Number 4, 1991 12.
- Iverson, K.E., *An Introduction to J*, Iverson Software Inc., 1992 1 8.
- Iverson, K.E., and E.E. McDonnell, *Phrasal Forms*, APL89, APL Quote-Quad, Volume 19, Number 4, 1989 8.
- Iverson, K.E., and A.T. Whitney, *Practical Uses of a Model of APL*, APL82, APL Quote-Quad, Volume 13, Number 1, 1982 9.
- Kernighan, B.W., and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
- McDonnell, E.E., and J.O. Shallit, *Extending APL to Infinity*, APL80, North-Holland Publishing Company, 1980.
- McIntyre, D.B., *Mastering J*, APL91, APL Quote-Quad, Volume 21, Number 4, 1991 8.
- McIntyre, D.B., *Language as an Intellectual Tool: From Hieroglyphics to APL*, IBM Systems Journal, Volume 30, Number 4, 1991 12.

Glossary and Index

An explanation is provided for every name in the program files. Each entry consists of a name, a program file name, a section number in this book (if any), and an explanation. The following conventions and abbreviations apply:

Names spelled with majuscules denote defined types (**typedefs**) or **#defined** constants and macros; those spelled with minuscules denote C functions and variables. Names localized in functions are omitted.

A sequence of letters; everywhere
Adverb
Argument
Character
Conjunction
Left noun argument to an adverb or a conjunction
Right noun argument to a conjunction
Left verb argument to an adverb or a conjunction
Right verb argument to a conjunction
Left argument
Right argument

| | | |
|----------|------|-------------------------------|
| a | * | left argument |
| A | j.h | 2.1 typedef array |
| AA | p.h | typedef transl. action |
| abase1 | ve.c | 8 monad #: |
| abase2 | ve.c | 8 dyad #: |
| ABS | j.h | absolute value |
| AC | j.h | 2.1 A reference count |
| ACTION | p.h | 1.2 parser action header |
| ac1 | au.c | 4 monad from C function |
| ac2 | au.c | 4 dyad from C function |
| ADERIV | a.h | 4 derive verb from adverb |
| adot1 | vp.c | 8 monad A. |
| adot2 | vp.c | 8 dyad A. |
| adv | p.c | 1.2 parser action |
| ADV | j.h | 2.2 A type |
| advform | ct.c | 1.3 derive an adv from a conj |
| aeq | vb.c | dyad = A subcase |
| AF | j.h | 3.1 typedef APL function |
| afi | vs.c | 8 dyad (standard index |
| aform | ct.c | bonded conjunction |
| afrom | vs.c | 8 dyad (A subcase |
| agenda | cg.c | 8 @. |
| AH | j.h | 2.1 A no. of header words |
| aii | u.c | no. of atoms in an item |
| all1 | u.c | 1 if all ones |
| alp | j.c | 8 a. |
| alt | aic | \$&1 _1@# |
| amp | c.c | 4 & |
| ampco | c.c | 8 &: |
| AN | j.h | 2.1 A number of atoms |
| ane | vh.c | fne A subcase |
| ANY | j.h | 2.2 A composite type |
| appf | xs.c | application file handler |
| appf1 | xs.c | appf subfunction |
| apv | u.c | 2.1 arithmetic progression |
| AR | j.h | 2.1 A rank |
| arep | r.c | 5.1 atomic representation |
| aro | r.c | 5.1 atomic rep, opened |
| arx | x.c | 5.1 monad 5!:1 |
| AS | j.h | 2.1 A shape |
| asgn | j.c | 6 last op was assignment |
| ASGN | j.h | 2.2 A type |
| ASSERT | j.h | 3.5 argument validation |
| ASSERTVV | c.c | 4 verb-verb case of conj |
| AS1 | a.h | 4 adverb-derived monad |

| | | | | | | | |
|---------|------|-----|------------------------|----------|-------|-----|------------------------|
| AS2 | a.h | 4 | adverb-derived dyad | | | | |
| AT | j.h | 2.1 | A type | C | j.h | 2.2 | typedef byte |
| atan2 | C | | | C* | j.c.h | 1.1 | character ID codes |
| atco | c.c | 8 | @: | ca | m.c | | copy array |
| atop | c.c | 8 | @ | CA | j.c.h | 1.1 | char type: letter |
| AV | j.h | 2.1 | A value | cadv | pc.c | 1.2 | :12 translator action |
| aojl | j.c | 2.1 | ojl | canta | vs.c | 8 | ! : subfunction |
| | | | | cantm | vs.c | | ! : on tables |
| B | j.h | 2.2 | typedef boolean | cant1 | vs.c | 8 | monad ! : |
| band | ve.c | | dyad *, B subcase | cant2 | vs.c | 8 | dyad ! : |
| base1 | ve.c | 8 | monad #. | car | m.c | | copy array recursively |
| base2 | ve.c | 8 | dyad #. | case | C | | |
| bdiv | ve.c | | dyad % B subcase | cases | p.c | 1.2 | parse table |
| behead | vs.c | 8 | monad !. | case1 | cg.c | 8 | monad m@.v |
| beq | vb.c | | dyad = B subcase | case2 | cg.c | 8 | dyad m@.v |
| BfromD | k.c | 7 | convert: B from D | catalog | vs.c | 8 | monad { |
| BfromI | k.c | | convert: B from I | catasp | xw.c | | , & ' ' |
| BfromZ | k.c | | convert: B from Z | CB | j.c.h | 1.1 | char type: B |
| bin | vm.c | | dyad ! subfunction | CC | j.c.h | 1.1 | char type: colon |
| binD | vm.c | | dyad ! subfunction | cconj | pc.c | 1.2 | :12 translator action |
| binI | vm.c | | dyad ! subfunction | ccurry | pc.c | 1.2 | :12 translator action |
| ble | vb.c | | dyad <: B subcase | ccvt | k.c | | convert: conditional |
| blt | vb.c | | dyad < B subcase | CD | j.c.h | 1.1 | char type: dot |
| bminus | ve.c | | dyad - B subcase | CDERIV | a.h | 4 | derive verb from conj |
| bool | a.c | 8 | b. | cdot1 | vp.c | 8 | monad C. |
| BOOL | j.h | 2.2 | A type | cdot2 | vp.c | 8 | dyad C. |
| booltab | a.c | 8 | function values for b. | cdyad | pc.c | 1.2 | :12 translator action |
| bool1 | a.c | | monad m b. | ceiling1 | ve.c | 8 | monad >. |
| bool2 | a.c | | dyad m b. | center | rl.c | 5.4 | S! : 4 subfunction |
| bor | ve.c | | dyad +. B subcase | ceg | vb.c | | dyad = C subcase |
| box | vs.c | 8 | monad < | cfd | vp.c | 8 | cycle from direct |
| BOX | j.h | 2.2 | A type | cforkv | pc.c | 1.2 | :12 translator action |
| boxq | x.c | 6.2 | monad 9! : 6 | cformo | pc.c | 1.2 | :12 translator action |
| boxs | x.c | 6.2 | monad 9! : 7 | cgaq | x.c | | PC monad 8! : 0 |
| bp | u.c | 2.2 | bytes per atom | cgas | x.c | | PC monad 8! : 1 |
| bplus | ve.c | | dyad + B subcase | cgav | x.c | | PC 8! : 0 setting |
| break | C | | | char | C | | |
| breaker | io.c | | check for user break | CHAR | j.h | 2.2 | A type |
| brem | ve.c | | dyad ! B subcase | chookv | pc.c | 1.2 | :12 translator action |
| bsdot | ap.c | 8 | \. | ciof | vh.c | 8 | dyad i. subfunction |
| bslash | ap.c | 8 | \ | circle | vm.c | 8 | dyad o. |
| BxD | k.c | | convert: B from D case | clock | C | | |
| BxI | k.c | | convert: B from I case | CLOCK* | j.h | | clock-related |
| BxZ | k.c | | convert: B from Z case | comonad | pc.c | 1.2 | :12 translator action |
| bytes | m.c | | bytes in use | CMPX | j.h | 2.2 | A type |

| | | | | | | | |
|----------|------|-----|-------------------------|---------|------|-----|---------------------------|
| CN | jc.h | 1.1 | char type: N | cut1 | cc.c | 8 | monad u; .n |
| coerce1 | u.c | 3.3 | coerce monad argument | cut2 | cc.c | 8 | dyad u; .n |
| coerce2 | u.c | 3.3 | coerce dyad arguments | CVCASE | k.c | | convert: case encoding |
| colon | cx.c | 8 | : | cvr | k.c | 2.2 | convert among B, I, D, Z |
| colorq | x.c | | PC monad 8!:4 | CX | jc.h | 1.1 | char type: other |
| colors | x.c | | PC monad 8!:5 | c2j | f.c | 6.3 | printf C to J format |
| compD | vg.c | | /: comparator | C9 | jc.h | 1.1 | char type: digit or sign |
| compI | vg.c | | /: comparator | | | | |
| compn | vg.c | | /: item size | D | jt.h | 2.2 | typedef real |
| compUC | vg.c | | /: comparator | dash | j.c | 2.1 | '--' |
| coninf | w.c | | ___ input handler | dbin | vm.c | | dyad ! D subcase |
| conj | p.c | 1.2 | parser action | ddiv | ve.c | | dyad % D subcase |
| CONJ | jt.h | 2.2 | A type | deal | v.c | 8 | dyad ? |
| CONJCASE | a.h | 4 | encode as NN,NV,VN,VV | DECLF | a.h | | declarations for x adv |
| conjug | ve.c | 8 | monad + | DECLFG | a.h | | declarations for x conj y |
| conname | w.c | | convert to name | decrem | ve.c | 8 | monad <: |
| connect | rt.c | 5.4 | 5!:4 subfunction | default | C | | |
| connum | w.c | | convert to numeric list | define | C | | |
| consa | pc.c | | :12 translator subfn | depth | p.c | | depth of function calls |
| onstr | w.c | | convert to string | deq | vb.c | | dyad = D subcase |
| continue | C | | | det | c.c | | monad u/ . v |
| on1 | cg.c | | monad m!:0 | dex | c.c | 8 |]. |
| on2 | cg.c | | dyad m!:0 | dexp | vm.c | | monad ^ D subcase |
| opy1 | xw.c | | monad 2!:4 | dfact | vm.c | | monad ! D subcase |
| opy1f | xw.c | | copy1 subfunction | dfc | vp.c | 8 | direct from cycle |
| opy2 | xw.c | | dyad 2!:4 | dfloor | ve.c | | monad <. D subcase |
| opy2f | xw.c | | copy2 subfunction | dfr | vp.c | 8 | direct from reduced |
| os | C | | | DfromZ | k.c | 7 | convert: D from Z |
| osh | C | | | dfs1 | p.c | | invoke named monad |
| PINF | f.c | 6 | printf _ | dfs2 | p.c | | invoke named dyad |
| PMINUS | f.c | 6 | printf minus sign | df1 | au.c | 4 | apply monad |
| PNAN | f.c | 6 | printf _. | DF1 | a.h | | derived monad header |
| PPLUS | f.c | 6 | printf plus sign | df2 | au.c | 4 | apply dyad |
| Q | jc.h | 1.1 | char type: quote | DF2 | a.h | | derived dyad header |
| S | jc.h | 1.1 | char type: space or tab | dgcd | ve.c | | dyad +. D subcase |
| str | u.c | 2.1 | C string into J string | dgrade1 | vg.c | 8 | monad \: |
| S1 | a.h | 4 | conj-derived monad | dgrade2 | vg.c | 8 | dyad \: |
| S2 | a.h | 4 | conj-derived dyad | divide | ve.c | 8 | dyad % |
| trans | pc.c | 1.2 | :12 translator | d1 | x.c | | monad 6!:3 |
| type | t.c | 1.1 | character type | d1cm | ve.c | | dyad *. D subcase |
| urry | p.c | 1.2 | parser action | d1e | vb.c | | dyad <: D subcase |
| urtail | vs.c | 8 | monad }: | d1t | vb.c | | dyad < D subcase |
| ut | cc.c | 8 | ; | dmin | ve.c | | dyad <. D subcase |
| ut01 | cc.c | 8 | monad v;:0 | dminus | ve.c | | dyad - D subcase |
| ut02 | cc.c | 8 | dyad v;:0 | dne | vh.c | | fne D subcase |

| | | | | | | | |
|---------|------|-----|--------------------------|---------|------|-----|----------------------|
| do | C | | | evger | cg.c | 8 | ": |
| DO | j.h | | do n times under index i | evmq | x.c | 3.5 | monad 91:8 |
| domerr | au.c | | verb with empty domain | evms | x.c | 3.5 | monad 91:9 |
| dot | c.c | 8 | . | evoke | u.c | | ! if in the form m~ |
| dotprod | c.c | 8 | dyad u/ . v | ex | x.c | | monad 41:55 |
| double | C | | | exec1 | v.c | 8 | monad " |
| dplus | ve.c | | dyad + D subcase | exec2 | v.c | 8 | dyad " |
| DR | cr.c | | derived rank | exit | C | | |
| dren | ve.c | | dyad D subcase | exp | C | | |
| drop | r.c | 5.2 | display representation | expn1 | vm.c | 8 | monad ^ |
| drop | vs.c | 8 | dyad). | expn2 | vm.c | 8 | dyad ^ |
| drx | r.c | 5.2 | 51:2 subfunction | extern | C | | |
| ds | x.c | 5.2 | monad 51:2 | | | | |
| ds | au.c | 3.1 | define symbol | fa | m.c | | free array |
| dsignum | ve.c | 7 | monad * D subcase | fabs | C | | |
| dtymes | ve.c | | dyad * D subcase | fac | vm.c | | monad ! subfunction |
| duble | ve.c | 8 | monad +: | fact | vm.c | 8 | monad ! |
| DxB | k.c | | convert: D from B case | factp1 | cf.c | 3.4 | monad ^! : n |
| DxI | k.c | | convert: D from I case | factp2 | cf.c | 3.4 | dyad ^! : n |
| DxZ | k.c | | convert: D from Z case | FAPPEND | x.h | | C file opcode |
| dyad | p.c | 1.2 | parser action | fclose | C | | |
| | | | | fdef | au.c | | derive verb/adv/conj |
| ebar | vb.c | 8 | dyad E. | feq | ut.c | 7 | equal within fuzz |
| EDGE | p.c | 1.2 | A composite type | ferror | C | | |
| edit | x.c | | PC monad 81:9 | fgetc | C | | |
| efr | cr.c | | effective rank | fgets | C | | |
| EI | w.c | 1.1 | word formation fn code | fh | vh.c | 8 | dyad i. hasher |
| eig1 | vm.c | 8 | monad c. | fi | u.c | | string to integer |
| eig2 | vm.c | 8 | dyad c. | fibon | cp.c | 8 | dyad u^: n |
| else | C | | | FILL | jl.h | | fill value |
| EN | w.c | 1.1 | word formation fn code | filler | u.c | | fill value |
| encell | f.c | 6.2 | boxed display subfn | FINDC | vh.c | 8 | dyad i. find in hash |
| endif | C | | | fit | cf.c | 3.4 | ! |
| enframe | f.c | 6.2 | boxed display subfn | fitct1 | cf.c | 3.4 | monad u! : n |
| ENGAP | f.c | | monad ": insert gap | fitct2 | cf.c | 3.4 | dyad u! : n |
| enstack | w.c | 1.1 | tokens subfunction | fitpp1 | cf.c | 3.4 | monad ":! : n |
| ensuite | cx.c | | \$. handler | fix | a.c | 8 | f. |
| eo | c.c | | u . . v and u . : v | fixa | a.c | | u f. subfunction |
| EPILOG | j.h | 2.3 | temps clean-up | fix1 | a.c | | u f. fn call depth |
| eps | vb.c | 8 | dyad e. | fixpath | a.c | | u f. fn call path |
| eq | vb.c | 8 | dyad = | fixpv | a.c | | u f. fixpath value |
| errsee | j.c | 3.5 | ! if display event msgs | FL | jl.h | 2.2 | A type |
| EV* | j.h | 3.5 | event codes | floor | C | | |
| even | c.c | 8 | .. | floor1 | ve.c | 8 | monad <. |
| every | a.c | | "each" operator variant | fmod | C | | |

| | | | | | | |
|--------|------|-----|---------------------------|---------|------|-------------------------------|
| fmtB | f.c | 6.1 | monad " : B subcase | GAPPEND | a.h | ` : opcode |
| fmtD | f.c | 6.1 | monad " : D subcase | gc | m.c | temps: purge; keep arg |
| fmtI | f.c | 6.1 | monad " : I subcase | gcd | ve.c | 8 dyad +. |
| fmtX | f.c | 6.1 | header for formatting fns | gc3 | m.c | temps: purge; keep args |
| fmtZ | f.c | 6.1 | monad " : Z subcase | ge | vb.c | 8 dyad >: |
| fn | vh.c | | 1 if items not equal | GGA | i.c | initialize constant |
| folk | ct.c | 1.3 | f g h | GG4 | i.c | initialize 4-byte constant |
| fontq | x.c | | MAC monad 8!:16 | GG8 | i.c | initialize 8-byte constant |
| fontS | x.c | | MAC monad 8!:17 | GINsert | a.h | ` : opcode |
| open | C | | | global | s.c | global symbol table |
| or | C | | | gn1 | xw.c | global names in 4!:1 |
| oreign | x.c | 8 | !: | grade | vg.c | /: subfunction |
| orko | ct.c | | size 3 a-train or c-train | grade1 | vg.c | 8 monad /: |
| orkv | p.c | 1.2 | parser action | grade2 | vg.c | 8 dyad /: |
| ork1 | ct.c | | monad f g h | graft | rt.c | 5.4 5!:4 subfunction |
| ork2 | ct.c | | dyad f g h | gt | vb.c | 8 dyad > |
| ormo | p.c | 1.2 | parser action | gtrain | ct.c | 1.3 m\ |
| putc | C | | | GTRAIN | a.h | 1.3 ` : opcode |
| puts | C | | | gt1 | pc.c | translator: :12 subfn |
| r | m.c | | free memory | gt2 | pc.c | translator: :12 subfn |
| ram | f.c | | boxed display subfn | | | |
| READ | x.h | | C file opcode | halve | ve.c | 8 monad -: |
| read | C | | | head | vs.c | 8 monad {. |
| real | ut.c | 7 | within fuzz of real | HOMO | jt.h | 2.1 1 if homogeneous |
| ree | C | | | hook | ct.c | 1.3 f g |
| REE | m.c | 2.3 | free cover | hooko | ct.c | 1.3 size 2 a-train or c-train |
| rom | vs.c | 8 | dyad { | hookv | p.c | 1.2 parser action |
| seek | C | | | hook1 | ct.c | monad f g |
| size | C | | | hook2 | ct.c | dyad f g |
| tell | C | | | host | xf.c | monad 0!:0 |
| UNC | jt.h | | A composite type | hostne | xf.c | monad 0!:1 |
| UPDATE | x.h | | C file opcode | htab | vh.c | dyad i. hash indices |
| WRITE | x.h | | C file opcode | hypoth | vz.c | complex: monad |
| write | C | | | | | |
| x | r.c | | 5!:1 inverse | I | jt.h | 2.2 typedef integer |
| xx | r.c | | 5!:1 inverse subfn | IC | jt.h | item count |
| xx | x.c | | 5!:0 | ID | jt.h | 3.1 ID field of verb/adv/conj |
| 1 | j.h | 3.1 | primitive monad header | iden | ai.c | 3.4 identity function adverb |
| 1RANK | j.h | 3.2 | monad rank handler | idiv | ve.c | dyad % I subcase |
| 2 | j.h | 3.1 | primitive dyad header | ieq | vb.c | dyad = I subcase |
| 2RANK | j.h | 3.2 | dyad rank handler | if | C | |
| | | | | ifdef | C | |
| a | m.c | | generate array | ifndef | C | |
| A | j.h | 2.1 | ga cover | ifrom | vs.c | 8 dyad { I subcase |
| am5243 | vm.c | | monad ! subfunction | IfromD | k.c | 7 convert: I from D |

| | | | | | | |
|---------|------|---------------------------|----------|------|-----|--------------------------|
| IfromZ | k.c | convert: I from Z | jc | x.c | C | LinkJ interface to C fn |
| igcd | ve.c | dyad +. I subcase | jc1r | vm.c | | dyad o. Z subcase |
| ii | u.c | 1.@# | jconjug | ve.c | | monad + Z subcase |
| iand | ap.c | 8 dyad u\ subfunction | jdiv | ve.c | | dyad % Z subcase |
| ilcm | ve.c | dyad *. I subcase | jdot1 | vm.c | 3.1 | monad j. |
| ile | vb.c | dyad <: I subcase | jdot2 | vm.c | 3.1 | dyad j. |
| ilt | vb.c | dyad < I subcase | jeq | vb.c | | dyad = Z subcase |
| imin | ve.c | dyad <. I subcase | jerr | j.c | 3.5 | error number |
| iminus | ve.c | dyad - I subcase | jexp | vm.c | | monad ^ Z subcase |
| immex | v.c | 0 immediate execution | JF | io.h | | typedef applic. file |
| immloop | j.c | 0 immex loop | jfappend | xf.c | | dyad 1!:3 |
| inbuf | io.c | user input buffer | jfdi | xf.c | | monad 1!:0 |
| include | C | | jferase | xf.c | | monad 1!:55 |
| incrm | ve.c | 8 monad >: | jfloor | ve.c | | monad <. Z subcase |
| indexof | vh.c | 8 dyad i. | jfopen | xf.c | | open file for processing |
| inf | j.c | _ | JFOPEN | io.h | | application file opcode |
| infile | j.c | input file handle | JFFRINT | io.h | | application file opcode |
| infix | ap.c | 8 dyad u\ | JFPROF | io.h | | application file opcode |
| inf1 | v.c | 8 monad _: | jfr | lj.c | C | LinkJ cover for fa |
| inf2 | v.c | 8 dyad _: | jfread | xf.c | | monad 1!:1 |
| initevm | i.c | initialize event msgs | JFSAVE | io.h | | application file opcode |
| insert | cg.c | monad m/ | jfsi | xf.c | | monad 1!:4 |
| INT | jt.h | 2.2 A type | jfwrite | xf.c | | dyad 1!:2 |
| int | C | | jgcd | ve.c | | dyad +. Z subcase |
| inv | ai.c | 3.4 ^:_1 | jgets | io.c | | get a line of user input |
| invamp | ai.c | 3.4 inv m&v u&n subcase | jinit | lj.c | C | LinkJ initializations |
| invdef | ai.c | 3.4 inv default | jinit2 | i.c | 0 | initializations |
| invl | cp.c | monad u^:_1 | jiread | xf.c | | monad 1!:11 |
| iota | v.c | 8 monad 1. | jiwrite | xf.c | | dyad 1!:12 |
| iplus | ve.c | dyad + I subcase | jlc | ve.c | | dyad *. Z subcase |
| ir | x.c | monad 3!:1 | jlog | vm.c | | dyad ^ Z subcase |
| irem | ve.c | dyad I subcase | jma | lj.c | C | LinkJ cover for ma |
| is | p.c | 1.2 parser action | jmag | ve.c | | monad Z subcase |
| isatty | C | | jminus | ve.c | | monad - Z subcase |
| isg | p.c | =: | jne | vh.c | | fine Z subcase |
| isignum | ve.c | monad * I subcase | joff | io.c | | sign-off |
| isl | p.c | =. | jot | j.c | 2.1 | <\$0 |
| IS1BYTE | jt.h | A composite type | jouts | io.c | | display a line |
| itake | vs.c | 8 dyad {, atomic left arg | jplus | ve.c | | dyad + Z subcase |
| itymes | ve.c | dyad * I subcase | jpow | vm.c | | dyad ^ Z subcase |
| IxB | k.c | convert: I from B case | jpr | f.c | 6 | display on screen |
| IxD | k.c | convert: I from D case | jputc | io.c | | display a character |
| IxZ | k.c | convert: I from Z case | jputs | io.c | | display a string |
| i0 | u.c | value of integer atom | jrem | ve.c | | dyad Z subcase |
| | | | jset | lj.c | C | LinkJ copula |

| | | | | | | | |
|----------|------|-----|---------------------------|----------|------|-----|--------------------------|
| signal | u.c | 3.5 | display event msg | lt | vb.c | 8 | dyad < |
| signum | ve.c | 7 | monad * Z subcase | | | | |
| SPR | j.h | | print short string | ma | m.c | | malloc cover |
| sqr | vm.c | | monad %: Z subcase | mag | ve.c | 8 | monad |
| stf | xs.c | | pointer to jstfrec | main | j.c | 0 | main() |
| stfrec | xs.c | | application file info | MALLOC | m.c | 2.3 | malloc cover |
| sti | io.h | | sesm input | malloc | C | | |
| stinit | io.h | | sesm initializations | mark | j.c | 1.2 | marker |
| stkiav | io.h | | sesm key input available | MARK | jt.h | 2.2 | A type |
| sto | io.h | | sesm output | mat | f.c | 6.2 | matrix image |
| stratts | x.c | | PC rd display attributes | match | vb.c | 8 | dyad -: |
| stref | x.c | | PC refresh display | math1 | vm.c | | math monad executor |
| stsatts | x.c | | PC set display attributes | math1z | vm.c | | math monad executor |
| stslow | x.c | | PC set slow display | math2 | vm.c | | math dyad executor |
| ststop | io.h | | sesm terminate | math2z | vm.c | | math dyad executor |
| tan2 | vz.c | | Atari GNU C kludge | matth1 | f.c | 6.2 | 1 14).@1_14).)@":@< |
| tymes | ve.c | | dyad * Z subcase | MAX | j.h | | maximum |
| x | lj.c | C | LinkJ execute | maxbytes | m.c | | max bytes used in line |
| | | | | maximum | ve.c | 8 | dyad >. |
| ey | ap.c | 8 | dyad u/. | mdiv | vi.c | 8 | dyad %. |
| F1 | k.c | | convert: function header | memchr | C | | |
| | | | | memcmp | C | | |
| | | | | memcpy | C | | |
| | | | | memset | C | | |
| _tmpnam | C | | | | | | |
| abel | cx.c | | m : n initialize labels | merge1 | a.c | 8 | monad u} |
| amin1 | vs.c | 8 | monad ,: | merge2 | a.c | 8 | dyad u) |
| amin2 | vs.c | 8 | dyad ,: | MIN | j.h | | minimum |
| c | u.c | | last character | minimum | ve.c | 8 | dyad <. |
| cm | ve.c | 8 | dyad *. | minors | c.c | 8 | monad u/ . v subfn |
| e | vb.c | 8 | dyad <: | minus | ve.c | 8 | dyad - |
| eft1 | v.c | 8 | monad [| minv | vi.c | 8 | monad %. |
| eft2 | v.c | 8 | dyad [| MMM | vz.c | | rotate z so that >:/+./z |
| ess | v.c | 8 | dyad -. | MOD2 | vm.c | | 2& |
| ev | c.c | 8 | [. | monad | p.c | 1.2 | parser action |
| ink | vs.c | 8 | dyad ; | move | p.c | 1.2 | parser action |
| INKJ | j.h | D | 1 if LinkJ | mr | u.c | | monadic rank |
| ocal | s.c | | local symbol table | mtv | j.c | 2.1 | \$0 |
| ocaltime | C | | | MTYOIN | io.h | | jsto opcode |
| og | C | | | MTYOUT | io.h | | jsto opcode |
| ogar1 | vm.c | 8 | monad ^. | mv | u.c | | move |
| ogar2 | vm.c | 8 | dyad ^. | mv1 | u.c | | move one atom |
| ong | C | | | | | | |
| ONG_MAX | j.h | | max I value | | | | |
| ONG_MIN | j.h | | min I value | NALP | j.h | | size of alphabet |
| PAR | jt.h | 2.2 | A type | NAME | jt.h | 2.2 | A type |
| r | u.c | | left rank | nan | j.c | | _. |

| | | | | | | |
|----------|------|-----|----------------------------|----------|------|------------------------------|
| nand | vb.c | 8 | dyad *: | obuf | j.c | buffer for short output |
| nc | s.c | | 4!:0 subfunction | obverse | c.c | 3.3 :. |
| ncases | p.c | 1.2 | no. of rows in cases | obv1 | c.c | monad u :. v |
| ncx | x.c | | monad 4!:0 | obv2 | c.c | dyad u :. v |
| NDEPTH | p.c | | max depth of fn calls | odd | c.c | 8 :. |
| ndig | w.c | | AMIGA LatticeC kludge | oind | ap.c | 8 monad u/. subfunction |
| ne | vb.c | 8 | dyad ~: | oldout | xs.c | old outfile value |
| negate | ve.c | 8 | monad - | omask | ap.c | 8 dyad u\ subfunction |
| negl | j.c | 2.1 | _1 | one | j.c | 2.1 1 |
| neg | vb.c | | dyad = subfunction | one1 | v.c | 8 monad 1: |
| NEVM | j.h | 3.5 | no. of event messages | one2 | v.c | 8 dyad 1: |
| NFKD | io.h | | size of fn key defn buffer | onm | s.c | opened name |
| NINB | io.h | | size of typeahead buffer | on1 | c.c | 4 monad u&v and u@v |
| NINPUT | j.h | | max length of input line | on2 | c.c | 4 dyad u&v |
| nla | s.c | | 4!:1 initials interest | ope | vs.c | 8 monad > |
| nline | cx.c | | m : n number of lines | oprod | a.c | dyad u/ |
| nmask | s.c | | 4!:1 numbers to type | ord | vp.c | 8 order of a permutation |
| NLOG | io.h | | size of session log | osub | ap.c | 8 monad u/. subfunction |
| nls | s.c | | 4!:1 subfunction | outfile | j.c | output file handle |
| nlx | s.c | | 4!:1 subfunction | outfix | ap.c | 8 dyad u\. |
| nll | x.c | | monad 4!:1 | outof | vm.c | 8 dyad ! |
| n12 | x.c | | dyad 4!:1 | over | vs.c | 8 dyad , |
| NMEM | m.c | | max size for malloc | OVERFLOW | j.h | large 0 value |
| NN | a.h | 4 | noun-noun case of conj | overr | vs.c | 8 dyad ,. |
| NOBUF | j.h | | length of obuf | | | |
| nor | vb.c | 8 | dyad +: | P | j.h | typedef primitives |
| norm | vi.c | | %:@(+/- . * +) | pad | rt.c | 5.4 5!:4 subfunction |
| not | ve.c | 8 | monad -. | parse | p.c | 1.2 interpret tokenized line |
| NOTCONJ | j.h | 1.2 | A composite type | pcopy1 | xw.c | monad 2!:5 |
| NOUN | j.h | 2.2 | A composite type | pcopy1f | xw.c | pcopy1 subfunction |
| NPP | j.h | | max value for qpp | pcopy2 | xw.c | dyad 2!:5 |
| NPROMPT | j.h | | max length of prompt | pcopy2f | xw.c | pcopy2 subfunction |
| NTH2 | f.c | 6.3 | dyad ": max width | pcvt | k.c | 2.2 convert if possible |
| NTSTACK | j.h | | temps: stack frame size | pdt | vi.c | dyad +/ . * |
| nu | w.c | 1.1 | national use alternatives | pfill | vp.c | 8 permutation fill |
| nub | v.c | 8 | monad ~. | PI | j.h | 3.14159265358 ... |
| nubsieve | vb.c | 8 | monad ~: | pie | j.c | 2.1 3.14159265358 ... |
| NUMERIC | j.h | 2.2 | A composite type | pind | vp.c | 8 positive indices |
| NV | a.h | 4 | noun-verb case of conj | pinv | vp.c | permutation inverse |
| NW | xw.c | | WS length of header | pix | vm.c | 8 monad o. |
| NWPFx | xw.c | | WS length of prefix | plus | ve.c | 8 dyad + |
| NWPtr | xw.c | | WS length of pointer | ply | cp.c | monad u^n |
| NXIL | xw.c | | WS block size in wcp | polar | vm.c | 8 monad *. |
| | | | | poly1 | vm.c | 8 monad p. |
| oblique | ap.c | 8 | monad u/. | poly2 | vm.c | 8 dyad p. |

| | | | | | | |
|----------|------|---------------------|---------|------|-----|-----------------------|
| povtake | v.c | monad > subfunction | razein | vb.c | 8 | monad e. |
| powop | cp.c | 8 | rbrace | a.c | 8 | } |
| prefix | ap.c | 8 | rc | f.c | 6.2 | boxed display subfn |
| PREF1 | a.h | 3.2 | rd | xf.c | | file read |
| PREF2 | a.h | 3.2 | rdot1 | vm.c | 8 | monad r. |
| prepare | cx.c | | rdot2 | vm.c | 8 | dyad r. |
| probe | s.c | | RE | j.h | 3.5 | return if error |
| prod | u.c | | recip | ve.c | 8 | monad % |
| PROLOG | j.h | 2.3 | rect | vm.c | 8 | monad +. |
| prompt | io.c | 0 | reduce | a.c | | monad u/ |
| promptq | x.c | | refresh | x.c | | PC monad 8!:7 |
| prompts | x.c | | reitem | vs.c | 8 | dyad \$ |
| prtscr | x.c | | repeat | vs.c | 8 | dyad # |
| ps | t.c | 3.1 | reshape | vs.c | | dyad \$. |
| psavel | xw.c | | residue | ve.c | 8 | dyad |
| psavelf | xw.c | | return | C | | |
| psave2 | xw.c | | reverse | vs.c | 8 | monad . |
| psave2f | xw.c | | rewind | C | | |
| psptr | t.c | 3.1 | rfd | vp.c | 8 | reduced from direct |
| ptr | f.c | 6.3 | RHS | jt.h | 1.2 | A composite type |
| PT | p.h | 1.2 | ri | x.c | | monad 3!:2 |
| ptr | jt.h | | right1 | v.c | 8 | monad) |
| punc | p.c | 1.2 | right2 | v.c | 8 | dyad) |
| | | | rin | vi.c | 8 | monad 128!:0 |
| rbx | j.c | 6.2 | rlq | x.c | | monad 9!:0 |
| rbt | cf.c | 7 | rls | x.c | | monad 9!:1 |
| rbvm | j.c | 3.5 | RMAX | j.h | | max rank |
| rbfuzz | j.c | 7 | RMAXL | j.h | | max rank as long |
| rbpp | cf.c | | roll | v.c | 8 | monad ? |
| rbprompt | j.c | | root | vm.c | 8 | dyad %: |
| rbq | c.c | 8 | ropen | xs.c | | open script |
| rbqr | vi.c | 8 | rotate | vs.c | 8 | dyad . |
| rbprl | j.c | 8 | round | vi.c | | <.0(0.56+) |
| rbpsort | C | | RPAR | jt.h | 2.2 | A type |
| | | | rr | u.c | | right rank |
| | | | RZ | j.h | 3.5 | return if 0 |
| | j.h | | | | | |
| | m.c | | | | | |
| rank | vs.c | 8 | S | jt.h | | typedef short integer |
| rankle | u.c | | SA | w.c | 1.1 | word formation state |
| rank1 | cr.c | | SAPPEND | x.h | | script opcode |
| ranklex | cr.c | | save1 | xw.c | | monad 2!:2 |
| rank2 | cr.c | | save1f | xw.c | | save1 subfunction |
| rank2ex | cr.c | | save2 | xw.c | | dyad 2!:2 |
| save1 | vs.c | 8 | save2f | xw.c | | save2 subfunction |
| save | v.c | 8 | sc | u.c | 2.1 | scalar integer |
| save | v.c | 8 | | | | |

| | | | | | | |
|----------|------|-----|--------------------------|----------|------|----------------------------|
| SCALARFN | a.h | | 1 if scalar function | sleep | C | |
| scalar4 | u.c | 2.1 | scalar 4-byte object | sl1_ | ai.c | u/"1" |
| scc | u.c | 2.1 | scalar character | SN | w.c | 1.1 word formation state |
| scf | u.c | 2.1 | scalar floating point | SNB | w.c | 1.1 word formation state |
| sclass | vb.c | 8 | monad = | SNZ | w.c | 1.1 word formation state |
| scnm | u.c | | scalar name | sp | x.c | monad 7! : 0 |
| scpt | xs.c | | scpt1/scpt2 subfn | spell | w.c | 1.1 spelling table |
| scpt1 | xs.c | | monad 0! : 2 and 0! : 3 | spellin | w.c | 1.1 ASCII string to ID |
| scpt2 | xs.c | | dyad 0! : 2 and 0! : 3 | spellout | w.c | 1.1 ID to string |
| script1 | xs.c | | monad 1! : 2 | spit | x.c | monad 7! : 2 |
| script2 | xs.c | | dyad 1! : 2 | sprintf | C | |
| SEEK_CUR | x.h | | fseek opcode | sp | x.c | monad 7! : 1 |
| SEEK_END | x.h | | fseek opcode | SQ | w.c | 1.1 word formation state |
| SEEK_SET | x.h | | fseek opcode | SQQ | w.c | 1.1 word formation state |
| seg | ap.c | 8 | monad u\ subfunction | sqrt | vm.c | 8 monad 3 : |
| self | * | 4 | A array for current verb | sqrt | C | |
| selfv | p.c | | old \$: value | square | ve.c | 8 monad * : |
| self1 | p.c | | monad \$: | SREAD | x.h | script opcode |
| self2 | p.c | | dyad \$: | srd | s.c | symbol table read |
| sesm | j.c | | 1 if using session mgr | srdlg | s.c | srd local or global |
| sesmexit | io.c | | session manager: epilog | sreduce | x.c | monad f/ for scalar f |
| sesminit | io.c | | session manager: prolog | srep | r.c | 5.3 string representation |
| sex | s.c | | symbol table expunge | srl | r.c | 5.3 srep subfunction |
| sex1 | cr.c | 3.3 | monad scalar executor | srx | x.c | 5.3 monad 5! : 3 |
| sex2 | cr.c | 3.3 | dyad scalar executor | SS | w.c | 1.1 word formation state |
| SF | j.h | 3.3 | typedef scalar function | sscript1 | xs.c | monad 1! : 3 |
| SF1 | v.h | 3.3 | scalar monad header | sscript2 | xs.c | dyad 1! : 3 |
| SF2 | v.h | 3.3 | scalar dyad header | ST | w.c | 1.1 typedef rhematic state |
| SGN | j.h | | signum | state | w.c | 1.1 rhematic state table |
| shape | vs.c | 8 | monad \$ | static | C | |
| shift1 | cf.c | 3.4 | monad 1! : n | stdin | C | |
| shift2 | cf.c | 3.4 | dyad 1! : n | stdnm | s.c | standardize name |
| shl | ai.c | | 1 & (1! : w) | stdout | C | |
| short | C | | | str | u.c | string of length n |
| shr | ai.c | | 1! : w | strcat | C | |
| sigflpe | i.c | | floating point exception | strchr | C | |
| sigint | i.c | | user interrupt | strcmp | C | |
| signal | C | | | strcpy | C | |
| signum | ve.c | 8 | monad * | strlen | C | |
| sin | C | | | strspn | C | |
| sinh | C | | | strtod | C | |
| size_t | C | | | struct | C | |
| sizeof | C | | | stype | x.c | 2.2 monad 3! : 0 |
| slash | a.c | 8 | / | sum | ve.c | monad +/ |
| sldot | ap.c | 8 | ./. | suffix | ap.c | 8 monad u\ |

| | | | | | | |
|----------|------|-----|----------------------------|----------|------|---------------------------|
| swap | a.c | 8 | ~ | tf | m.c | temps: free old frame |
| swap1 | a.c | | monad u~ | tfail | pv.c | translator: 1 if failed |
| swap2 | a.c | | dyad u~ | tfloor | ut.c | 7 tolerant <. |
| switch | C | | | tg | m.c | temps: get new frame |
| SX | w.c | 1.1 | word formation state | th | f.c | 6.1 monad " : num subcase |
| SY | jt.h | 2.2 | typedef symb tab entry | thbox | f.c | 6.2 monad " : A subcase |
| SYMB | jt.h | 2.2 | A type | thn | vh.c | 8 dyad i. hash bytes |
| symbolis | s.c | | symbol table set | thorn1 | f.c | 6 monad " : |
| symbrd | s.c | | symbol table read | thorn2 | f.c | 6.3 dyad " : |
| SYS | j.h | D | system ID | ths | vh.c | 8 dyad i. start of hash |
| SYS_* | j.h | D | system IDs and masks | tie | cg.c | 8 |
| system | C | | | time | C | |
| SZ | w.c | 1.1 | word formation state | time_t | C | |
| S9 | w.c | 1.1 | word formation state | t1e | ut.c | 7 tolerant <: |
| | | | | t1eaf | rl.c | 5.4 5! : 4 subfunction |
| taa | ct.c | 1.3 | trains: adv adv | t1t | ut.c | 7 tolerant < |
| TAA | ct.c | 1.3 | trains: adv adv case | tmpnam | C | |
| taaa | ct.c | 1.3 | trains: adv adv adv | tname | pv.c | translator: names |
| TAAA | ct.c | 1.3 | trains: adv adv adv case | tokens | w.c | 1.1 build parse stack |
| TAAc | ct.c | 1.3 | trains: adv adv conj case | tostdout | j.c | 1 if output to stdout |
| table | vs.c | 8 | monad , . | totbytes | m.c | bytes used in session |
| tac | ct.c | 1.3 | trains: adv conj | tparse | pv.c | translator: :11 and :12 |
| TAC | ct.c | 1.3 | trains: adv conj case | tpop | m.c | 0 temps: purge |
| taca | ct.c | 1.3 | trains: adv conj adv | tpush | m.c | temps: insert new entry |
| TACA | ct.c | 1.3 | trains: adv conj adv case | traverse | m.c | apply t1r to each atom |
| tacc | ct.c | 1.3 | trains: adv conj conj | treach | rl.c | 5.4 |
| TACC | ct.c | 1.3 | trains: adv conj conj case | trep | rl.c | 5.4 tree representation |
| TACT | p.h | 1.2 | translator action header | troot | rl.c | 5.4 5! : 4 subfunction |
| tail | vs.c | 8 | monad { : | trr | rl.c | 5.4 5! : 4 subfunction |
| take | vs.c | 8 | dyad { . | trx | x.c | monad 5! : 4 |
| tally | vs.c | 8 | monad # | ts | x.c | monad 6! : 0 |
| tbase | m.c | | temps: base | tsit | x.c | monad 6! : 2 |
| tca | ct.c | 1.3 | trains: conj adv | tss | x.c | monad 6! : 1 |
| TCA | ct.c | 1.3 | trains: conj adv | tssbase | j.c | time base |
| tcaa | ct.c | 1.3 | trains: conj adv adv | tstack | m.c | temps (A*) AV(tstacka) |
| TCAA | ct.c | 1.3 | trains: conj adv adv | tstacka | m.c | temps current stackframe |
| TCAC | ct.c | 1.3 | trains: conj adv conj | ttokens | pv.c | translator: tokenize |
| TCC | ct.c | 1.3 | trains: conj conj | ttop | m.c | 2.3 temps top |
| tcca | ct.c | 1.3 | trains: conj conj adv | tval | pv.c | translator: values |
| TCCA | ct.c | 1.3 | trains: conj conj adv | twprimes | s.c | sizes for symbol tables |
| tccc | ct.c | 1.3 | trains: conj conj conj | two | j.c | 2.1 2 |
| TCCC | ct.c | 1.3 | trains: conj conj conj | tymes | ve.c | 8 dyad * |
| ceil | ut.c | 7 | tolerant >. | typedef | C | |
| DECL | ct.c | | trains: declarations | | | |
| eq | ut.c | 7 | tolerant = | U | jt.h | unsigned |

| | | | | | | |
|-----------|------|-----------------------------|---------|------|-----|--------------------------|
| UC | j.h | typedef unsigned byte | WB | f.c | 6.1 | monad " : B max width |
| under | c.c | 8 | wcp | xw.c | | WS copy |
| UNDERFLOW | j.h | small D value | WD | f.c | 6.1 | monad " : D max width |
| under1 | c.c | monad u&.v | wend | xw.c | | WS offset to directory |
| under2 | c.c | dyad u&.v | wex | xw.c | | WS expunge |
| unlink | C | | wexf | xw.c | | WS wex subfunction |
| ung | xw.c | WS remove given names | WF1 | xw.c | | WS monad header |
| unquote | p.c | monad or dyad m~ | WF2 | xw.c | | WS dyad header |
| unquo1 | a.c | monad m~ | while | C | | |
| unquo2 | a.c | dyad m~ | while1 | cp.c | | monad u^:v |
| unsigned | C | | while2 | cp.c | | dyad u^:v |
| unsr | r.c | 5! : 3 inverse | WI | f.c | 6.1 | monad " : I max width |
| until | C | | with1 | c.c | 4 | monad m&v |
| unw | r.c | unsr subfunction | withr | c.c | 4 | monad u&n |
| upon2 | c.c | dyad u&v | wnc | xw.c | | WS name class |
| | | | wncf | xw.c | | WS wnc subfunction |
| V | j.h | 4 typedef verb | wnl | xw.c | | WS name list |
| vadv | pv.c | 1.2 : 11 translator action | wnlf | xw.c | | WS wnl subfunction |
| VAV | j.h | 2.1 AV for verb/adverb/conj | wopen | xw.c | | WS open |
| vconj | pv.c | 1.2 : 11 translator action | wopr1 | xw.c | | WS monad executor |
| vcurry | pv.c | 1.2 : 11 translator action | wopr2 | xw.c | | WS dyad executor |
| vdya | pv.c | 1.2 : 11 translator action | wordil | w.c | 1.1 | word index and length |
| VERB | j.h | 2.2 A type | words | w.c | 1.1 | monad " : |
| vfin | xf.c | validate file name | wp | xw.c | | WS dir names |
| vforkv | pv.c | 1.2 : 11 translator action | WP | j.h | | words per array |
| vformo | pv.c | 1.2 : 11 translator action | wpfx | xw.c | | WS prefix |
| vhookv | pv.c | 1.2 : 11 translator action | wptr | xw.c | | WS wp and wq location |
| vi | u.c | validate integer | wq | xw.c | | WS dir index/length/type |
| vib | u.c | validate integer, bounded | wr | io.c | | write to screen |
| vis | pv.c | 1.2 : 11 translator action | wmdir | xw.c | | WS write directory |
| vmonad | pv.c | 1.2 : 11 translator action | WREAD | x.h | | WS opcode |
| vmove | pv.c | 1.2 : 11 translator action | wtype | t.c | | ctype clone for wordi |
| vn | u.c | validate noun | WUPDATE | x.h | | WS opcode |
| VN | a.h | 4 verb-noun case of conj | WWRITE | x.h | | WS opcode |
| vnm | s.c | validate name | WZ | f.c | 6.1 | monad " : Z max width |
| void | C | | | | | |
| vpunc | pv.c | 1.2 : 11 translator action | xadv | cx.c | | x m : 1 |
| vs | u.c | validate string | XC | x.h | | ! : argument encoding |
| vtrans | pv.c | 1.2 : 11 translator | xconj | cx.c | | x m : 2 y |
| VV | a.h | 4 verb-verb case of conj | xcv | k.c | 2.2 | convert to "lowest" type |
| v2 | u.c | integer pair | xcvta | k.c | | xcvt subfunction |
| | | | xd | cx.c | | monad or dyad m : n |
| w | * | right argument | xdash | r.c | 5.4 | "dash" box drawing char |
| wa | xf.c | file write or append | xdged | ve.c | | dyad +. D subcase subf |
| WATERLOO | j.h | D 1 if Waterloo libraries | xdrem | ve.c | | dyad D subcase subf |

| | | | | | |
|----------|----------|---------------------------|---------|--------|-------------------------|
| xigcd | ve.c | dyad +. I subcase subfn | zmag | vz.c 8 | complex: monad 1 |
| xil | xw.c | WS dir, index/length | zminus | vz.c | complex: dyad - |
| XINF | j.h | _ internal representation | zmj | vz.c | complex: 0j_1 |
| xirem | ve.c | dyad 1 I subcase subfn | zm4 | vz.c 8 | complex: _4&o. |
| XNAN | j.h | _ internal representation | znegate | vz.c | complex: monad - |
| xnl | cx.c | monad m : y | znoncel | vz.c | complex: nonce error |
| xn2 | cx.c | dyad x : n | znonce2 | vz.c | complex: nonce error |
| xvl | cx.c | monad u : y | ZNZ | vz.c | complex: 1 if nonzero |
| xv2 | cx.c | dyad x : v | ZOV | vz.c | complex: 1 if overflow |
| z | * | result | zplus | vz.c | complex: dyad + |
| Z | jt.h 2.2 | typedef complex | zpow | vz.c | complex: dyad ^ |
| zacos | vz.c 8 | complex: _2&o. | zp4 | vz.c 8 | complex: 4&o. |
| zacosh | vz.c 8 | complex: _6&o. | zp8 | vz.c 8 | complex: 8&o. |
| zarc | vz.c 8 | complex: 12&o. | zrem | vz.c | complex: dyad 1 |
| zasin | vz.c 8 | complex: _1&o. | zsin | vz.c 8 | complex: 1&o. |
| zasin | vz.c 8 | complex: _5&o. | zsinh | vz.c 8 | complex: 5&o. |
| ZASSERT | vz.c | complex: arg validation | zsqr | vz.c | complex: monad %: |
| zatan | vz.c 8 | complex: _3&o. | ZS1 | vz.c | complex: monad shell |
| zatanh | vz.c 8 | complex: _7&o. | ZS2 | vz.c | complex: dyad shell |
| zceiling | vz.c | complex: monad >. | ztan | vz.c 8 | complex: 3&o. |
| zcir | vz.c 8 | complex: dyad o. | ztanh | vz.c 8 | complex: 7&o. |
| zconj | vz.c | complex: monad + | ztrend | vz.c 8 | complex: monad * |
| zcos | vz.c 8 | complex: 2&o. | ztypes | vz.c | complex: dyad * |
| zcosh | vz.c 8 | complex: 6&o. | ZUN | vz.c | complex: 1 if underflow |
| zdiv | vz.c | complex: dyad % | ZxB | k.c | convert: Z from B case |
| ZEPILOG | vz.c | complex: standard exit | ZxD | k.c | convert: Z from D case |
| zeq | vz.c | complex: dyad = | ZxI | k.c | convert: Z from I case |
| ZEQ | vz.c | complex: 1 if equal | z1 | vz.c | complex: 1j0 |
| zero | j.c 2.1 | 0 | | | |
| zeroZ | j.c | 8-byte zero | | | |
| zerol | v.c 8 | monad 0: | | | |
| zero2 | v.c 8 | dyad 0: | | | |
| zexp | vz.c 8 | complex: monad ^ | | | |
| ZEZ | vz.c | complex: 1 if zero | | | |
| zfloor | vz.c 8 | complex: monad <. | | | |
| ZF1 | vz.c | complex: monad header | | | |
| ZF1DECL | vz.c | complex: declarations | | | |
| ZF2 | vz.c | complex: dyad header | | | |
| ZF2DECL | vz.c | complex: declarations | | | |
| zgcd | vz.c | complex: dyad +. | | | |
| zgcd1 | vz.c | complex: zgcd subfn | | | |
| zj | vz.c | complex: 0j1 | | | |
| zlem | vz.c | complex: dyad *. | | | |
| zlog | vz.c 8 | complex: monad ^. | | | |

Appendix E. Foreign Conjunction

x, xf, xs, etc. are names of C program files

| | | | |
|-------|---------------------------|--------|-------------|
| 0!:0 | host xf * | 6!:0 | ts x * |
| 0!:1 | hostne xf * | 6!:1 | tss x * |
| 0!:2 | script1 xs * script2 xs | 6!:2 | tsit x * |
| 0!:3 | sscript1 xs * sscript2 xs | 6!:3 | dl x * |
| 0!:55 | joff u * | | |
| | | 7!:0 | sp x * |
| 1!:0 | jfdir xf * | 7!:1 | sps x * |
| 1!:1 | jfread xf * | 7!:2 | spit x * |
| 1!:2 | * jfwrite xf | | |
| 1!:3 | * jfappend xf | 8!:0 | cgaq x * |
| 1!:4 | jfsize xf * | 8!:1 | cgas x * |
| 1!:11 | jiread xf * | 8!:4 | colorq x * |
| 1!:12 | * jiwrite xf | 8!:5 | colors x * |
| 1!:55 | jferase xf * | 8!:7 | refresh x * |
| | | 8!:9 | edit x * |
| 2!:0 | * wnc xw | 8!:16 | fontq x * |
| 2!:1 | wml xw * | 8!:17 | fonts x * |
| 2!:2 | savel xw * save2 xw | 8!:19 | prtscr x * |
| 2!:3 | psavel xw * psave2 xw | | |
| 2!:4 | copy1 xw * copy2 xw | 9!:0 | rlq x * |
| 2!:5 | pcopy1 xw * pcopy2 xw | 9!:1 | rls x * |
| 2!:55 | * wex xw | 9!:4 | promptq x * |
| | | 9!:5 | prompts x * |
| 3!:0 | stype x * | 9!:6 | boxq x * |
| 3!:1 | ir x * | 9!:7 | boxs x * |
| 3!:2 | ri x * | 9!:8 | evmq x * |
| | | 9!:9 | evms x * |
| 4!:0 | ncx : * | | |
| 4!:1 | nl1 : * nl2 : | 10!: | jc x * |
| 4!:55 | ex : * | | |
| | | 128!: | qr vi * |
| 5!:0 | fx x * | 128!:1 | xinv vi * |
| 5!:1 | ar x * | | |
| 5!:2 | dr x * | | |
| 5!:3 | sr x * | | |
| 5!:4 | tr x * | | |

Appendix F. System Summary

vb, p, v, etc. are names of C program files

| | | | |
|----|----------------------|------------------------|------------------------|
| = | sclass vb * eq vb | isl p | isg p |
| < | box v * lt vb | floor1 ve * minimum ve | decrem ve * le vb |
| > | ope v * gt vb | ceil1 ve * maximum ve | increm ve * ge vb |
| - | coninf w | coninf w | infl v * inf2 v |
| + | conjug ve * plus ve | rect vm * gcd ve | duble ve * nor vb |
| * | signum ve * times ve | polar vm * lcm ve | square ve * nand vb |
| - | negate ve * minus ve | not ve * less v | halve ve * match vb |
| % | recip ve * divide ve | minv vi * mdiv vi | sqroot vm * root vm |
| ^ | expn1 vm * expn2 vm | logar1 vm * logar2 vm | * powop cp |
| \$ | shape vs * reitem vs | ensuite cx | self1 p * self2 p |
| ~ | swap s * | nub v * | nubslave vb * ne vb |
| | mag ve * residue ve | reverse vs * rotate vs | cant1 vs * cant2 vs |
| . | * dot c | * even c | * odd c |
| : | * colon cx | * obverse c | |
| / | ravel vs * over vs | table vs * overr vs | lamin1 vs * lamin2 vs |
| : | rave v * link v | * cut cc | words w * |
| # | tally vs * repeat vs | basel ve * base2 ve | abasel ve * abase2 ve |
| ! | fact vm * outof vm | * fit cf | * foreign x |
| / | slash s * | sldot ap * | gradel v * grade2 v |
| \ | bslash ap * | bsdor ap * | dgradel v * dgrade2 v |
| { | left1 v * left2 v | * lev c | |
| } | right1 v * right2 v | * dex c | |
| { | catalog vs * from vs | head vs * take vs | tail vs * |
| } | rbrace s * | behead vs * drop vs | curtail vs * |
| " | * qq cr | exec1 v * exec2 v | thorn1 f * thorn2 f |
| ' | * tie cg | | * dvger cg |
| @ | * atop c | * agenda cg | * atco c |
| & | * amp c | * under c | * ampco c |
| ? | roll v * deal v | | |
|) | label cx | a. alp j | A. adot1 vp * adot2 vp |
| b. | bool s * | c. eig1 vm * eig2 vm | C. cdot1 vp * cdot2 vp |
| e. | ravein vb * eps vb | E. * ebar vb | f. fix s * |
| i. | iota v * indexof vh | j. jdot1 vm * jdot2 vm | NB. wordil w |
| o. | pix vm * circle vm | p. poly1 vm * poly2 vm | r. rdot1 vm * rdot2 vm |
| x. | xd cx | y. xd cx | 0: zero1 v * zero2 v |
| | | | 1: one1 v * one2 v |